

1N-93-CR

0 CIT

26411

69P

NASW-4801
Final Report¹

Non-Dipolar Magnetic Field Models
and Patterns of Radio Emission:
Uranus and Neptune Compared

N95-12469

Unclass

G3/93 0026411

Dr. D. R. Evans
Radiophysics, Inc.
5475 Western Ave.
Boulder CO 80301

(NASA-CR-196412) NON-DIPOLAR
MAGNETIC FIELD MODELS AND PATTERNS
OF RADIO EMISSION: URANUS AND
NEPTUNE COMPARED Final Report, 3
Aug. 1993 - 15 Sep. 1994
(Radiophysics) 69 p

¹ A condensed version of this report is to be submitted to *Geophysical Research Letters*

ABSTRACT

The magnetic field geometries of Uranus and Neptune are superficially similar, and are similarly unlike those of other planets: the field strengths are similar, and they contain extraordinarily large non-dipolar components. As a corollary, the best dipolar field models of each of the two planets comprises a dipole that is considerably offset from the planetary centre and tilted away from the rotational axis.

However, in other respects the best field models of the two planets are quite different. Uranus has a quadrupole model in which all the terms are well determined and in which none of the higher order terms is determined. To represent the magnetometer data acquired during Voyager's Neptune encounter requires a model of order 8 (instead of Uranus' order 2), yet many of the coefficients are poorly determined. A second model, an octupole model comprising the terms up to order three of the order 8 model, has been suggested by the magnetometer team as being useful; its use, however, is limited only to the region outside of about $2 R_N$, whereas planetary radio emissions have their sources well inside this surface.

Computer code has been written that permits an analysis of the detailed motion of low energy charged particles moving in general planetary magnetic fields. At Uranus, this code reveals the existence of an isolated region of the inner magnetosphere above the day side in which particles may be trapped, separate from the more general magnetospheric trapping. An examination of the so-call ordinary mode uranian radio emissions leads us to believe that these emissions are in fact extraordinary mode emissions coming from particles trapped in this isolated region.

A similar attempt to discover trapping regions at Neptune has proved, unfortunately, to be impossible. This arises from three factors: a) the computation needed to track particles in an eighth order field is more than an order of magnitude greater than that needed to perform a similar calculation in a quadrupole field, and is beyond the capacity of workstation-class computers; b) the octupole field model is known to be in error by too large an amount for it, or any similarly truncated version of the eighth order model, to produce trustworthy results; c) the eighth order model can, in effect, be infinitely varied without affecting the field strength along the spacecraft trajectory.

1 -- MOTION OF CHARGED PARTICLES

Decametric (and longer wavelength) radio emissions from the vicinity of a planet are caused by the acceleration of non-relativistic charged particles in the planetary magnetosphere. The exact mechanism by which most such emission is created is still a subject of some debate, although most planetary radio astronomers currently believe that some variant of the cyclotron maser mechanism¹ is responsible. A good review of many of the competing theories is given in chapter 9 of Dessler (1983).

One common feature of the mechanisms is that they are all *coherent*. This requirement is demanded by virtue of the high intensities observed in planetary radio emissions. Because of this coherence, and also the remarkable degree of repeatability of most such emissions, it appears likely that the great majority of emissions come from coherently coupled groups of particles trapped inside planetary magnetospheres.

1.1 -- MOTION IN A PLANETARY MAGNETIC FIELD

A charged particle emits radiation whenever it undergoes an acceleration; that is, whenever it is subject to a force. In general, both electrical and magnetic forces may act on a charged particle (we ignore collisions here and throughout this discussion except where we specifically state otherwise). The equation of motion of a charged particle subject to an electrical field $\mathbf{E}$ and a magnetic field $\mathbf{B}$ is given by the deceptively simple equation:

$$\mathbf{F} = q\mathbf{E} + q(\mathbf{v} \times \mathbf{B})$$

where: $\mathbf{F}$ is the force experienced by the particle;

q is the particle's charge;

$\mathbf{v}$ is the particle's velocity.

For the remainder of our discussion, we will assume that the electrical field is zero, and hence the force on the particle is simply:

$$\mathbf{F} = q(\mathbf{v} \times \mathbf{B})$$

The ramifications of this simple equation are discussed at length in a number of elementary textbooks (such as Roederer, 1970) and will only briefly be reviewed here.

A particle moving in a general magnetic field can experience three levels of periodicity in its motion, if only average particle positions are considered. (That is, if one is not interested in the very small scale, precise position of the particle.) At the lowest, most rapid, level, is the particle's cyclotron motion, the periodicity of the motion perpendicular to the local magnetic field. At an intermediate level is the possibility of a bounce motion, along a line of force. At the highest, slowest level, is the particle's drift motion, along a closed surface of field lines and instantaneously perpendicular to those field lines. We say that a particle is trapped if it exhibits all three kinds of motion. To put it another way, once a particle finds itself in a state in which it exhibits all three periodic kinds of motion, it will continue to traverse the same magnetic field surface unless something external affects the particle's motion. (Such as, for example, a collision with another particle, or energy loss through radiation.)

¹ See, for example, Wu and Lee (1979).

Not all magnetic field topologies permit the trapping of particles, and, of those that do, not all particles moving in the field will be trapped. A dipole magnetic field, which is, on the largest scale, the appearance of a typical planetary field, does permit such trapping.

1.2 -- DIPOLE MAGNETIC FIELDS

It is easy to show that for a particle moving in a conservative magnetic field, the magnetic moment of the particle,

$$\mu = (\sin^2 \alpha / |\mathbf{B}|)$$

(where α , the angle between the particle's velocity and the local magnetic field, is called the pitch angle) is a constant of the motion.

In a dipole field, a particle initially starting with a pitch angle different from 90° will travel along a field line, its pitch angle changing continuously, until it reaches a point (providing that the planet's body, a satellite, or an ionosphere does not intervene) where the strength of the local field constrains its pitch angle to be zero. At this point there is a Lorentz force parallel to the field line in such a direction that the particle moves back in the direction it has come. Thus a bouncing motion is set up, symmetrical about the magnetic equator.

As the particle bounces along a field line, the curvature of the magnetic field also induces a slower drift in a direction perpendicular to the field line. In the case of a purely dipolar field, a charged particle simultaneously performs rapid gyro movement, a slower bounce up and down field lines, always mirroring at the same latitude (north and south) and slowly drifts around the field. The locus of this motion traces out a symmetric surface in space. This surface is known as an L shell. It has an associated number, given by the ratio of the distance of the particle from the planet's centre at the point that it crosses the magnetic equator to the planetary radius. (That is, L shells always have values greater than unity.)

1.3 -- NON-DIPOLAR MAGNETIC FIELDS

The simple situation described above can become considerably more complex when charged particles move in planetary fields that are non dipolar. One basic condition still holds: if a particle exhibits motion that returns it to its starting point, it is trapped. In the dipole case, all trapping regions are essentially similar: they are symmetrical about the equator and rotationally symmetric about the axis of the dipole. Neither of these is generally true when the field is non-dipolar.

Non-dipolar magnetic fields are usually specified by the terms of a spherical harmonic expansion of the form:

$$V = a \sum_{n=1}^{\infty} \left\{ \left(\frac{r}{a}\right)^n T_n^e + \left(\frac{a}{r}\right)^{n+1} T_n^i \right\}$$

where V is the scalar potential from which $\mathbf{B}$ can be derived ($\mathbf{B} = -\nabla V$);

a is the equatorial radius of the planet;

and $T_n^i = \sum_{m=0}^n \{ P_n^m(\cos \theta) [g_n^m \cos(m\phi) + h_n^m \sin(m\phi)] \};$

The T_n^e represent external currents and so can be ignored. The $P_n^m \cos \theta$ are Schmidt normalised associated Legendre functions of degree n and order m , and the g_n^m and the h_n^m are the Schmidt coefficients. (For a considerably more detailed discussion, see the earlier monthly reports).

There is (in general) no way in which a mere inspection of the terms of such an expansion can provide insight into the existence and topography of trapping regions within that field. The only guaranteed way in which to show the existence of such regions is to search exhaustively for them by means of a computer program.

The writing of such a program is nontrivial. Many of the assumptions and simplifications that are valid in the dipole case are not so for non-dipolar fields (although it is often not obvious why this is so). Consequently, particle motion, except for the very smallest level of motion, the gyro motion, needs to be calculated and followed to determine whether, eventually, the particle returns to its initial starting point.

The author has written such a program, and it is provided in Listing I. Ancillary programs are provided in Listings II and III.

The basic functioning of the program can be quite simply described. Input comprises the latitude, longitude and pitch angle of a particle (assumed to be a 1 eV electron). The program considers a point on the surface of the planet (making allowance for oblateness) described by the latitude and longitude. The field line that intersects the planetary surface at that point is then traced until a point is reached (the first such point) where the field strength is at a local minimum. The particle is moved to that point, provided with the given pitch angle and 1 eV of energy, and then released. The program follows the particle's subsequent motion.

The motion may be quite complex. In particular, two facts lead to the likelihood of increased complexity over the simple dipolar case: the non monotonic decrease of field strength along a radius and the presence of multiple mirror equators. We discuss these two items separately.

1.3.1 -- THE DECREASE OF FIELD STRENGTH WITH DISTANCE

It is tempting to extrapolate the dipole case, where moving out along a radius leads to a monotonically decreasing local field, to the non-dipolar case; but to do so would be an error. Indeed, it is possible to construct entirely feasible planetary field models at which the field strength some distance above the surface of the planet goes to zero.

How this can come about can be visualised quite simply. Consider a field model with only two components, one dipolar and one of considerably higher order. Physically, every coefficient in the spherical harmonic expansion corresponds to one or more current loops beneath the planet's surface. The higher the order represented by the coefficient, the closer to the surface its equivalent current loops lie. One can easily imagine a pair of current loops of opposite sign, with the smaller valued current lying closer to the surface. Now, since one loop is located at the centre of the planet and one loop some distance towards the surface, it is quite clear that the field strength at and above the surface near the second current loop will be lower than at other points on the

surface. If the second loop is sufficiently close to the surface, it is even possible that the magnetic field will reverse in direction at the surface, being dominated by the field from the second, closer loop. Yet, far from the planet along the same radius, the field from the larger loop at the planet's centre will dominate. Ergo, there must be some point in between at which the two fields exactly cancel out.

This is not simply a theoretical exercise. Figure 1 shows a slice through the neptunian I8E1 44ev model of Connerney *et al.*, 1991. The slice begins at latitude 30° , longitude 210° (measured in the positive ϕ direction) and ends at latitude -60° , longitude 330° . The strength of the model field from the surface to a distance of 1 radius above the surface is shown. Two distinct regions of minimal field strength are obvious. The lowest of them drops very close (perhaps all the way) to zero.

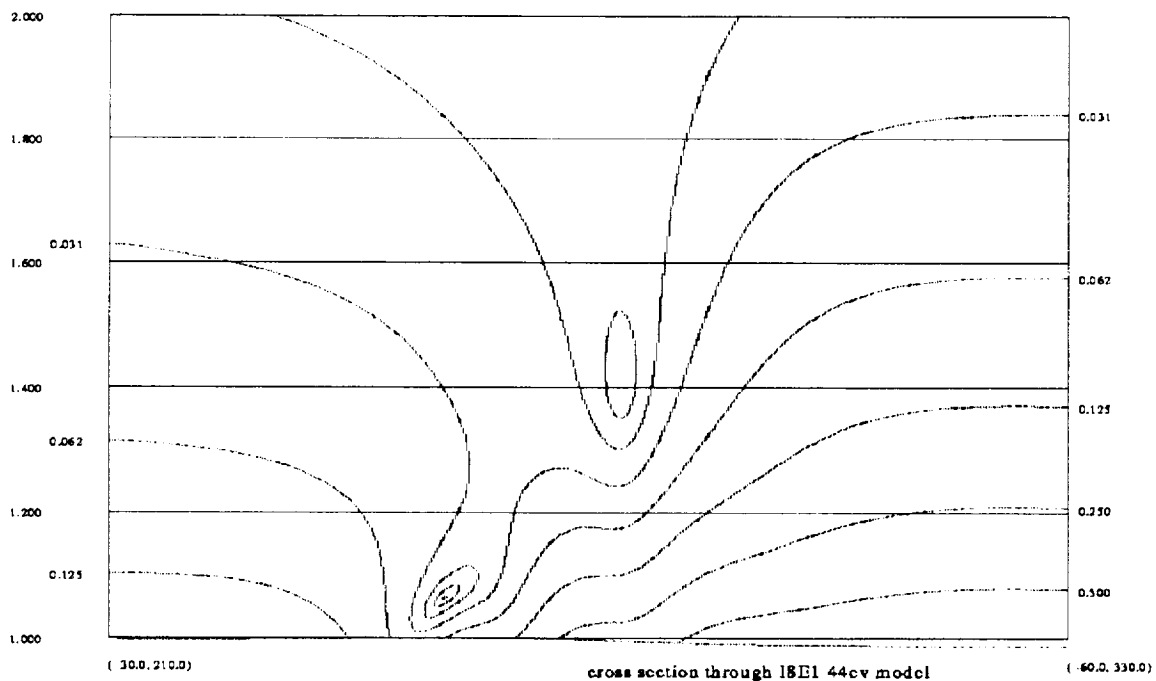


Figure 1

Such figures make it clear why simplistic notions such as L shells and monotonically decreasing field intensity with distance are invalid in general, non-dipolar, planetary magnetic fields.

1.3.2 -- MULTIPLE MIRROR EQUATORS

We define the term "mirror equator" as follows: a mirror equator is the locus of points of local minimum of field strength along adjacent field lines.

In a dipole field, there are an infinite number of mirror equators, all confined to (and defining) a sheet perpendicular to the dipole axis and passing through the equator. Any given field line intersects exactly one mirror equator, at the dipole's equator, and the field line is always perpendicular to the mirror equator..

In a non-dipolar field, none of these is necessarily true. Because of the relatively convoluted shape of non-dipolar fields, it is frequently the case that a single field line exhibits more than one local minimum along its length; the mirror equator corresponding to that minimum may be a considerable distance from any plausible definition of the field's equator¹; and the field line may intersect the equator non-perpendicularly.

In a dipolar field, all mirror equators are necessarily closed lines; in fact, they are constrained to be circles about the dipole axis. Neither of these constraints exists for non-dipolar fields. One can easily imagine a case in which a particular field line exhibits a minimum along its length at a particular point, but an adjacent field line shows no such minimum. Thus, mirror equators in non-dipolar fields may have ends that simply float in space. They may also intersect the planet's body, leading to ends of a different kind..

Mirror equators are fundamental to the problem of trapping, for a trapped particle must necessarily pass through a mirror equator on every bounce. It is at that point that its pitch angle is a maximum. We note that that maximum pitch angle may vary as it slowly drifts across adjacent field lines, and that because of the conservation of magnetic moment, a particle's pitch angle may determine whether a particular mirror equator appears as a closed loop to that particle. Bifurcations in mirror equators are also possible, and the precise path taken by a charged particle at such points is dependent on its energy, pitch angle and even the direction in which it is travelling along the field line. Such minutiae are unmodellable with any degree of rigour and, in any case, are well beyond the confidence with which we should hold non-terrestrial planetary magnetic field models. We therefore simply regard particles that travel close to the ends or bifurcations of mirror equators as untrapped.

For a particle to be trapped within a particular region of the magnetosphere, there must exist a closed mirror equator within that region. If such an equator exists, then a low energy particle with a pitch angle at the mirror equator sufficiently close to 90° will be trapped.

A particle may pass through a mirror equator caused by a local minimum in field strength but have a pitch angle sufficiently different from 90° that it can pass into a different region of the magnetosphere than one with a pitch angle closer to 90° . Therefore, in non-dipolar fields, loss cones come about not merely because of the planet's surface, but also because of the detailed topography of the magnetic field.

¹ In general, the notion of an "equator", as indeed any notion of magnetic latitude or longitude, is invalid in a non-dipolar field; unfortunately, this rather obvious fact is frequently ignored in the literature.

Thus, in searching for possible trapping regions (which really becomes a search for closed mirror equators), it behooves one to specify an initial pitch angle, at a mirror equator, close to 90° , as particles with pitch angles much different may not be trapped at all. For our computer simulations, then, we typically chose a pitch angle of 89° . Of course, unless the pitch angle of real trapped particles can range to quite different values, over say several tens of degrees, the number of particles trapped is unlikely to be sufficient to support detectable emission. However, after a closed mirror equator has been detected in a model field, one can then vary the pitch angle accordingly to determine the robustness of the mirror equator.

2 -- URANUS

The best uranian magnetic field model is the Q_3 model of Connerney *et al.* (1987). This is a quadrupole (i.e. order 2) magnetic field model, and, as such, displays relatively little complexity. We note that the actual field at Uranus may exhibit substantial higher order terms. The Q_3 model is simply the most exiguous model that fits the Voyager magnetometer data to within the instrumental errors. A closer flyby (or a more accurate instrument) could well have resolved higher order moments.

The Q_3 model is essentially dipolar, especially in the nightside hemisphere. The relatively high values of the quadrupole coefficients come about because of the large offset and tilt of the essentially dipolar field. (The first uranian magnetic field model (of Ness *et al.*, 1986) was an offset, tilted dipole model in which the dipole was offset by approximately $0.3 R_U$ from the planetary centre, and tilted some 60° from the planet's axis of rotation.; the axis of this dipole intersected the planet at $+15.2^\circ$, $47.7^\circ W$ and -44.2° , $227.5^\circ W$.)

In the dayside hemisphere, the Q_3 model has an anomolous, secondary surface dip equator bounding the area roughly from 290° to 60° longitude and 35° to 85° latitude (see Figure 7 of Connerney *et al.*, 1987). Figure 2 is a view of the night side of Uranus, with its essentially dipolar field. Figure 3 is a similar view of the day side, and the fact that the dayside field is not particularly dipolar is quite obvious. The driver computer programs used to produce these Figures is provided in Listing IV¹.

In reference to their Figure 7, Connerney *et al.* (1987) refer to the secondary dip equator and correctly state (in different words) that this divides the inner magnetosphere into two distinct, separate regions, in the smaller of which "field lines are trapped close to the planet's surface and are essentially isolated from the rest of the magnetosphere". They fail, however, to draw the corollary: that this permits the possibility of a second magnetic equator, and thus a separate and distinct region of the magnetosphere in which particles might be trapped .

(Note that the presence of a second dip equator does not *imply* a second trapping region; it merely *permits* such a region.)

¹ For the sake of brevity, the ancillary programs, which run to approximately two thousand lines of code, are not included in this report.

134.20, 132.50

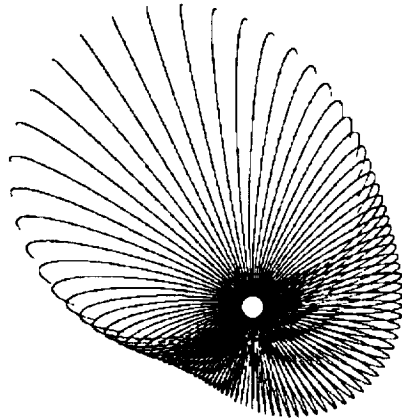


Figure 2

74.80, 312.30

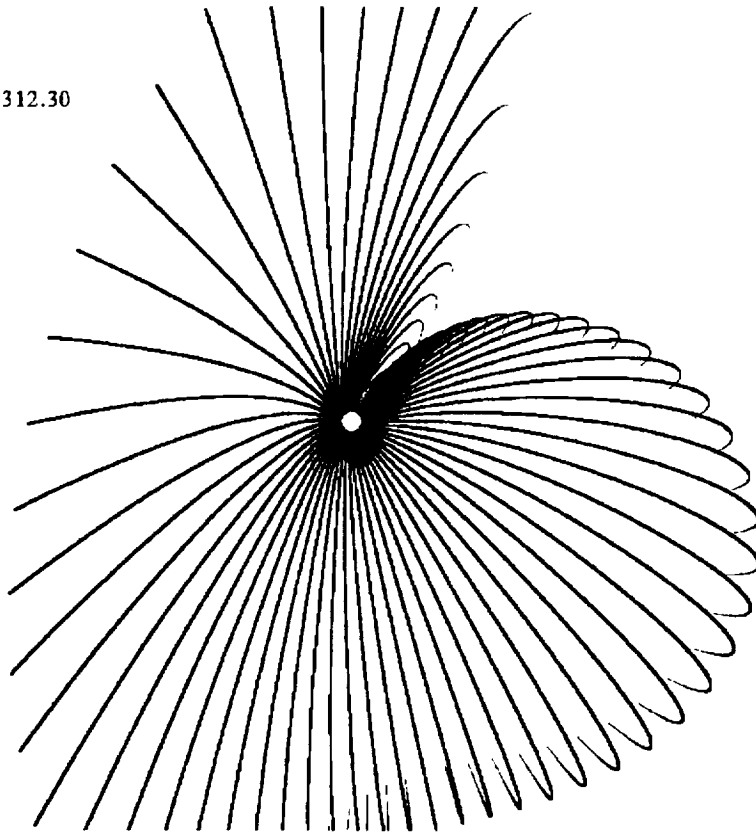


Figure 3

In order to determine whether a second trapping region actually exists, an exhaustive search of the space surrounding the planet must be undertaken. Such a search involves constructing a fine

meshed grid and permitting low energy particles with pitch angles near 90° to move according to the laws of motion described above.

A search of this nature requires a prodigious quantity of computation. The search can be rendered somewhat more tractable by limiting oneself to a curved two dimensional grid (the surface of the planet) rather than the entire three dimensional grid in which the planet is enmeshed. Such a simplification does not materially affect the fineness of the search, because in the three dimensional case many of the points lie on or near field lines on which other points on the mesh lie. Using the surface of the planet eliminates this quasi-redundancy.

The procedure followed, then, was to cover the planet with a fine grid of points, and then, for each such point to "walk" down the associated field line until a minimum (not necessarily a global minimum) was reached. At that point a particle was released with an 89° pitch angle, and its motion was followed. The minimum of the field line is on a closed mirror equator if the particle returns to its original position.

This process consumed a large amount of computer time. Most grid points led to the dipole mirror equator, of course. However, a second mirror equator, associated with the anomalous dayside region, was also located. The surface boundary of this equator is an elongated oval, approximately bounded in latitude by 39° and 42° and in longitude by 60° to 130° . This equator, in the Q_3 model, is barely a closed loop for particles with a pitch angle of 89° . The three dimensional area in which particles with smaller pitch angles are trapped is limited to less than 10° in longitude at the surface. However, the mirror equator barely drops into the ionosphere of the planet, and only a very small change in the model (corresponding to a small internal current loop near the surface) would cause the equator to be closed over a considerably wider range of pitch angles. Such a change would be undetectable at the distance at which Voyager flew past the planet and would have no effect on particle motion in the remainder of the magnetosphere.

Figure 4, which is an overhead view looking down at Uranus from latitude 40° and longitude 60° , shows this region. The two dark areas are trapped regions for particles with pitch angles of 89° . The curved hook represents the paths of typical particles with an equatorial pitch angle of 70° emanating from the heart of the trapped region. Particles with pitch angles less than this collide with the planet and are not trapped.

40.00, 60.00

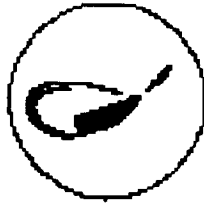


Figure 4

Prior to the Uranus encounter, low frequency (< 100 kHz) radio emissions were detected by the Voyager Planetary Radio Astronomy (PRA) instrument. These were quite different from the much stronger nightside emissions detected at and subsequent to closest approach (at 1800 hours on day 24 of 1986) which corresponded to particles moving in the larger cavity of the magnetosphere. Desch and Kaiser (1987), relying on the fact that this emission was observed to be left hand polarized, have posited that it is ordinary mode emission created "via a direct generation process" of which no details are given. Desch and Kaiser used an offset, tilted dipole model (in which, of course, there is no possibility of a secondary mirror equator) and they state that the consequences of the Q_3 model "have not been investigated" but nevertheless opine that their conclusion "seems firm".

However, propagating ordinary mode emissions have never been observed coming from a giant planet, and there are good theoretical reasons for believing that such emissions are, at best, extremely difficult to produce in nature. Additionally, their conclusion rests solely on purely dipolar notions of trapping and mirroring, in the very region in which the uranian magnetosphere is *not* dipolar. Given the results of our search for a secondary mirror equator, a much more likely scenario presents itself: the dayside emissions detected by PRA prior to the uranian encounter are the result of particles trapped in a small region on the dayside. The polarity of the local field is opposite to that of the main field in the surrounding regions, and so left hand polarised emissions coming from such a region are *extraordinary* mode (that is, in the somewhat confusing terminology of magneto-ionic theory, they are the usual mode of excitations of planetary radio emissions).

Kaiser and Desch go to considerable length to separate their putative ordinary mode emission from the more general, lower frequency so-called "smooth narrow band" emissions¹. Figure 5 shows the PRA data from midnight to 1600 hours on day 24. According to Kaiser and Desch, the emission in channel 194 from approximately 0300 to approximately 0800 is *not* the same as that in channel 195, a mere 19 kHz (or 15% in frequency) away. Figure 5 indicates that there is no

¹ A name which is misleading, since they are certainly not narrow band.

particular reason to make this distinction.¹ Further, they lend great weight to the observed time of commencement (but not cessation, which does not fit their model particularly well) of the emissions. However, the field line geometry predicted by the Q_3 model is complex, and the relative geometry of the spacecraft and the trapping region is changing rapidly throughout this period, so it is eminently reasonable to expect different aspects of the same emission to make themselves visible over these hours.

However, even if one (unnecessarily) concedes the notion that the higher frequency emission is intrinsically different from the lower frequency, the weight of evidence suggests that it is extraordinary mode emission coming from a small, isolated region of the inner magnetosphere around 40° latitude, 60° longitude, rather than some exotic ordinary mode emission associated with a dipole model that is known to be incorrect in the very region in question.

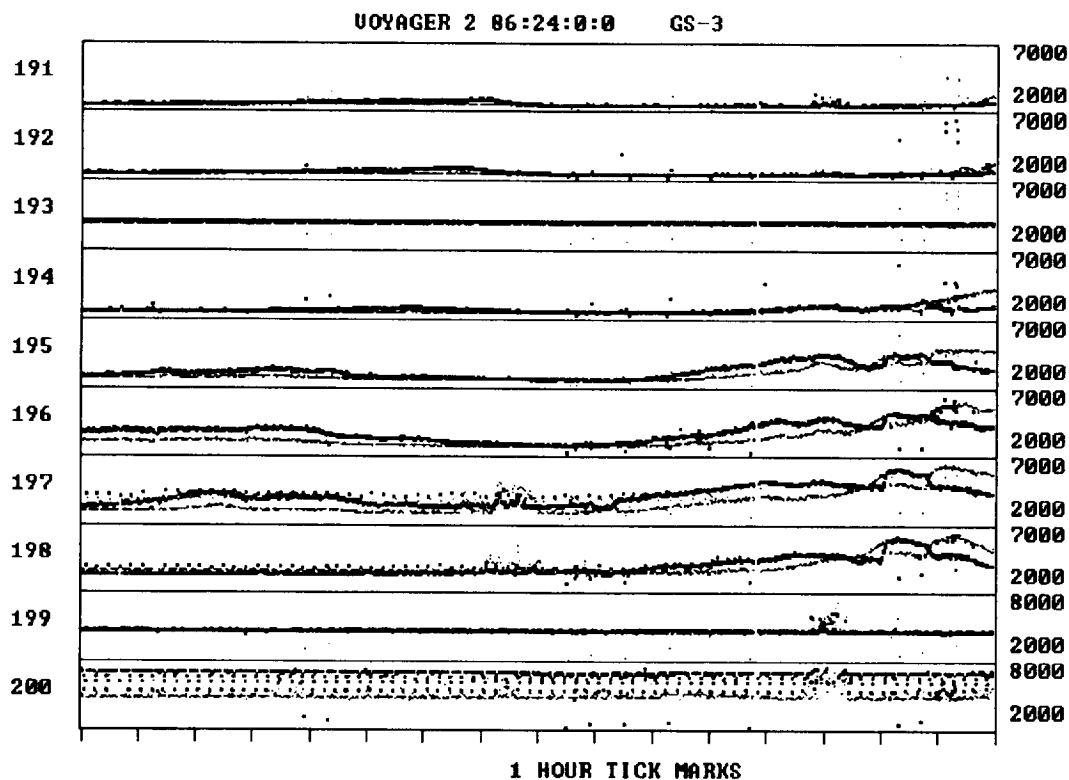


Figure 5

¹ Their strongest "evidence" comes from a measurement of the ellipticity of the emissions seen below channel 195 as compared to that seen above that channel; however they make no attempt to examine the instrumental response at different frequencies to signals with known ellipticity.

3 -- NEPTUNE

The situation at Neptune is vastly more complicated than is that at Uranus. Much of this added complexity is a result merely of the greatly different encounter geometry, and is (probably) not intrinsic.

At Uranus, the encounter geometry was such that all the dipole and quadrupole coefficients of the spherical harmonic expansion could be extracted from the observational dataset with relative ease. Further, the spacecraft's closest approach to Uranus was sufficiently distant that no coefficients of higher order could be reliably determined. Consequently, the Voyager magnetometer team presented the community with a relatively "clean" magnetic field model of order two, in which higher order coefficients were undetermined. The situation at Neptune was quite different.

At Neptune, the encounter geometry was such that the spacecraft entered and exited the system from the southern hemisphere; but, during the actual encounter period, the spacecraft flew extremely close to the northern pole of the planet (closest approach was a mere $1.18 R_N$, as opposed to a closest approach distance of $4.2 R_U$ at Uranus. This resulted in great difficulties for the magnetometer team in producing the best field model (and, moreso, great conceptual difficulties for those trying to use the model provided by Connerney *et al.*; the literature contains several papers that show that neither their authors nor the papers' referees gave Connerney *et al.*'s paper the careful reading it deserves and requires).

Because of the peculiar nature of the neptunian encounter geometry, the paper presenting the magnetic field models is lengthy and must be read very carefully. The paper is impossible to *précis* here, and a discussion has already been given in a prior monthly report, so we shall here merely summarize the points that are most important for our purposes.

Connerney *et al.* took the magnetometer data and then systematically inverted the data to extract the coefficients of a spherical harmonic expansion, limiting the order to some number N . They then compared the results of the ensuing N^{th} order model to the observed data. They repeated this process, gradually increasing N , until they reached a point at which the model field matched the observed field to within the known accuracy of the magnetometer. This required a much higher order model than for any of the other giant planets (because the close flyby permitted higher order moments to be detected). However, many coefficients (corresponding to fields in the southern hemisphere) were poorly constrained (if they were constrained at all). Thus the model that they obtained and published, the I8E1 44ev model, while sufficient to reproduce the field along the spacecraft's trajectory, could be (and almost certainly is) in considerable error at distances far from the trajectory. In addition, the I8E1 44ev model suffers from the fact that because it is an 8th order model, it requires a large number of relatively complicated calculations to be made in order to calculate the field strength at a single point. In this regard it is more than ten times as complex as the uranian quadrupole field¹.

¹ See the program MAGFIELD.CC and its associated header file, MAGFIELD.H, in Listing II.

Because of these difficulties, Connerney *et al.* proposed a second model, the O_8 model, which is a pure octupole model. However, this model has severe restrictions, which are described in the paper but have generally been ignored by the community. In the first place, the O_8 model is not in any sense the *best* octupole model of the neptunian field. It is not formed by inverting the data and constraining all coefficients higher than order three to be zero (which is the procedure used to minimise the difference between model predictions and the observed data). Rather, it is simply the I8E1 44ev model *truncated* at order 3, which is quite a different matter. Additionally, Connerney *et al.* show clearly that the O_8 model, while predicting field strength outside $2 R_N$ quite well, is not to be relied on inside that distance. Low energy trapped radiation, of course, is almost guaranteed to come from well inside $2 R_N$.

It is tempting to do as other authors have done and to simply use the O_8 model for calculations. However, a simple comparison of the two models show that this would be inappropriate.

On a global scale, the two models appear little different. Figures 6 and 7 show "wire model" views looking at the northern and southern axis of the neptunian dipole model¹ using the O_8 model; Figures 8 and 9 show the same views, but with the I8E1 44ev model.

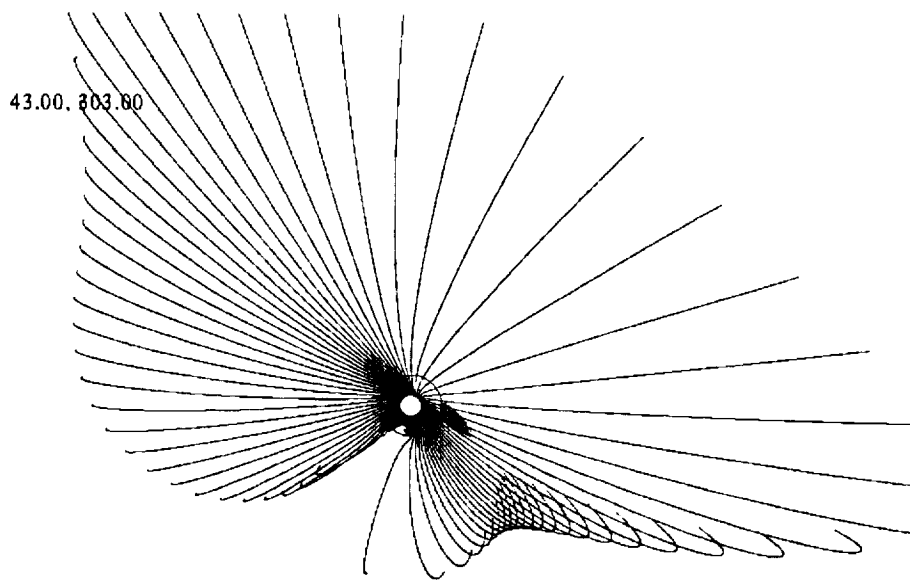


Figure 6

¹ This dipole is offset from the planetary centre by $0.55 R_N$ and has a tilt of 46.8° . For more details, see Ness *et al.*, 1989.

125.00, 82.00

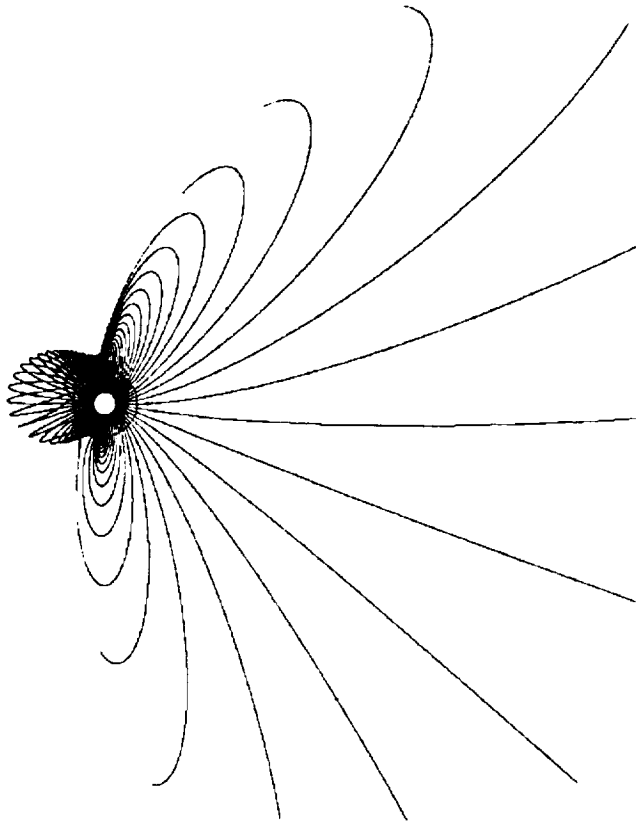


Figure 7

43.00, 303.60

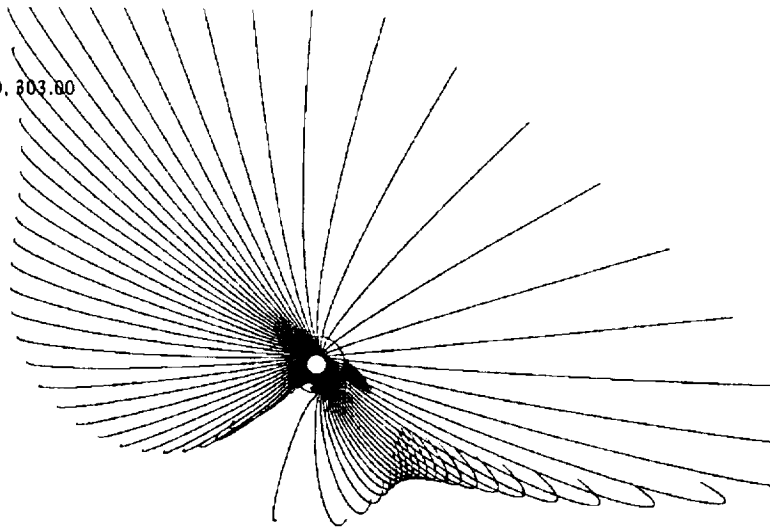


Figure 8

125.00, 82.00

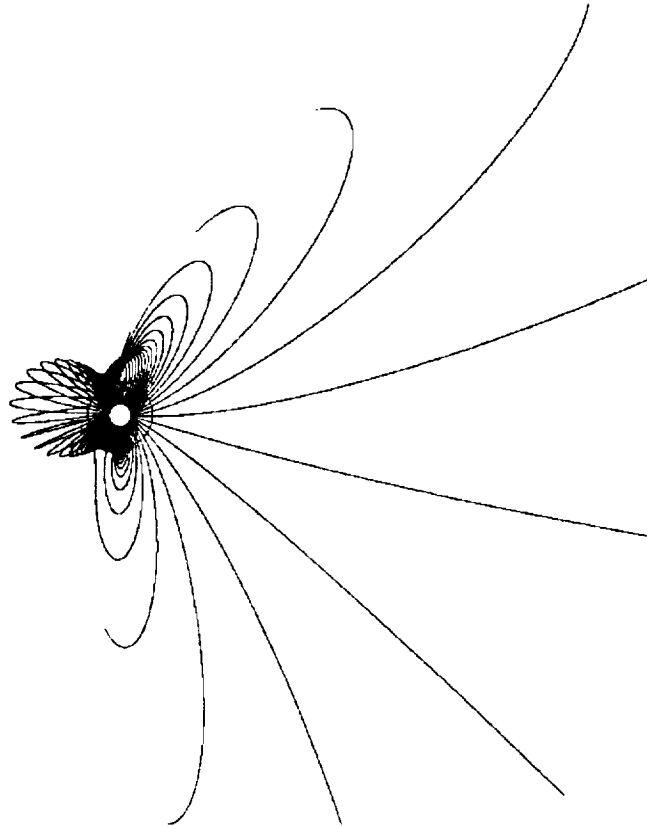


Figure 9

There is almost no discernible difference between these pictures except in the details in the inner magnetosphere. However, one must remember that up to and including order three, the models are *identical*. Therefore it should come as no surprise that global views such as these, which are dominated by the low order terms, appear almost the same. A better understanding of the difference between the models is provided by Figures 10 through 15. These show the ratios of the two models at a series of distances above the planet's surface. As is to be expected, in general (with one exception), the further one rises in altitude, the more similar the two models become. This is simply a reflection of the fact that in a spherical harmonic expansion, higher order terms can be thought of as resulting from smaller current loops closer to the planet's surface. Thus, higher order terms tend to have a more local effect than lower order terms. At altitudes below about $1.6 R_N$, the difference between the two models is generally several tens of percent, and sometimes even more.

Figure 16 shows even more clearly the necessity of using the most accurate field model for purposes of searching the inner magnetosphere for possible trapping regions. It shows the same cross section as does Figure 1, except for the O_8 model instead of Figure 1's I8E1 44ev model. The contours close to the surface are completely different in the two cases. In particular, the above-ground decrease in magnetic field intensity that was so much a feature of Figure 1 is completely absent in Figure 16. Thus, regions in which trapping might appear in the I8E1 44ev model will be completely absent of such trapping in the O_8 model.

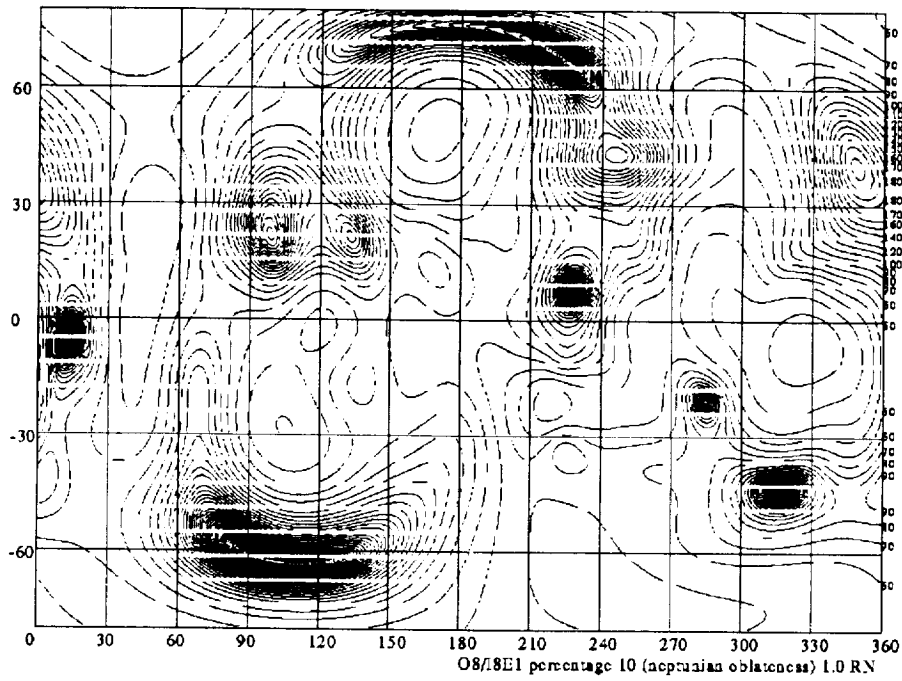


Figure 10

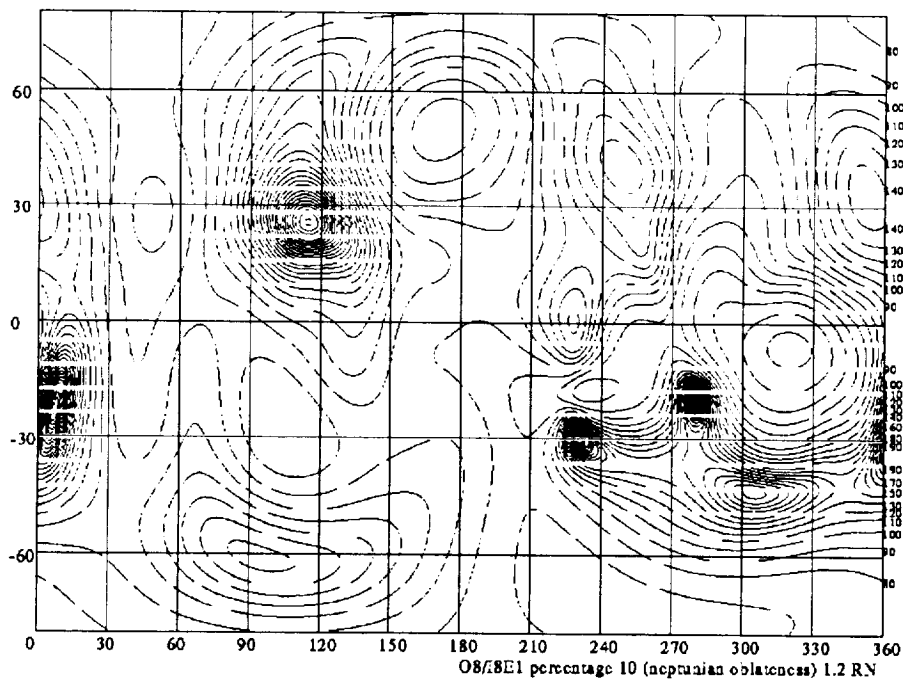


Figure 11

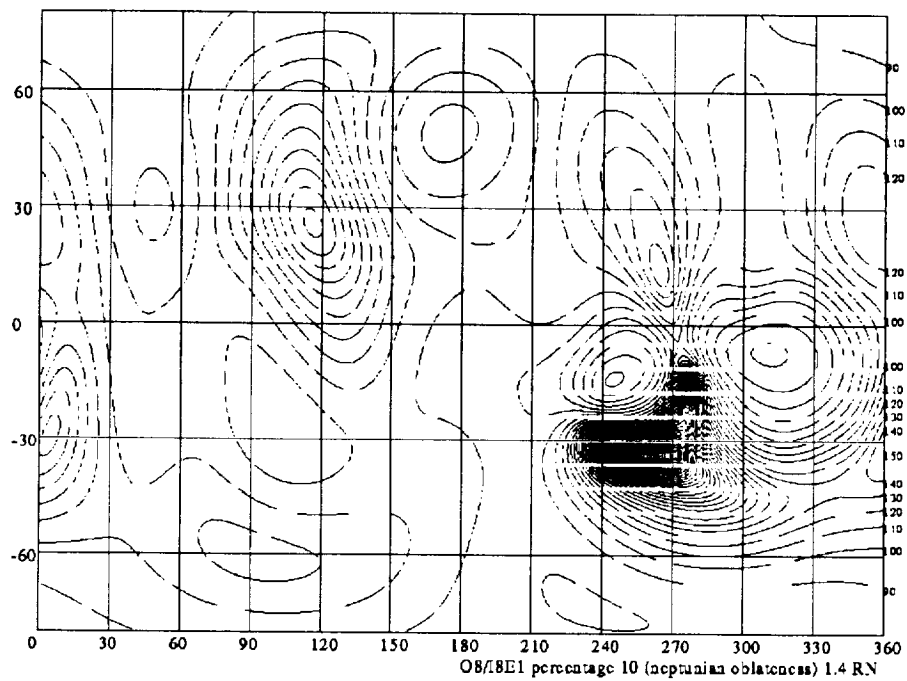


Figure 12

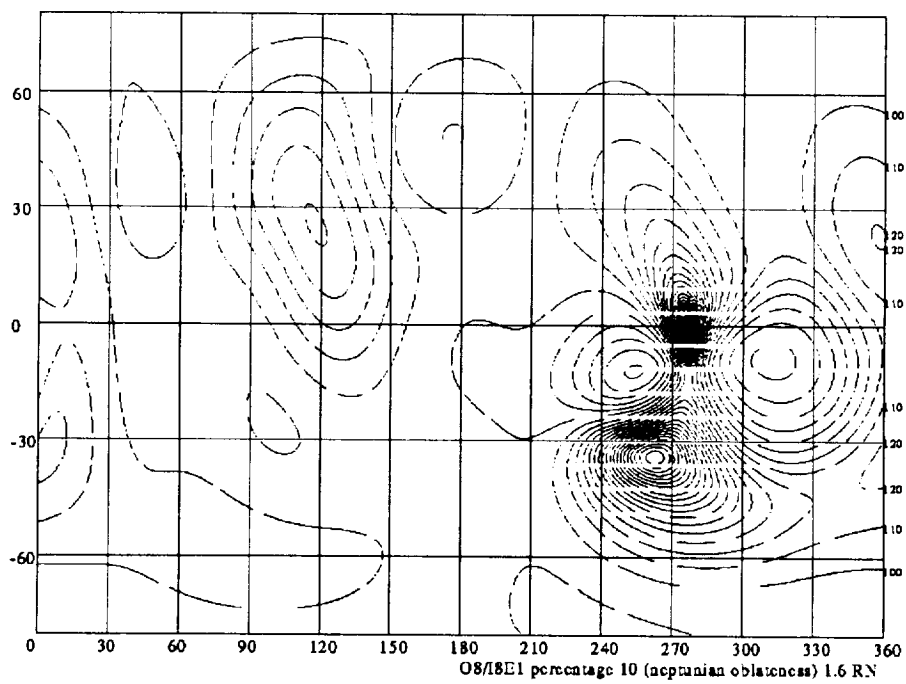


Figure 13

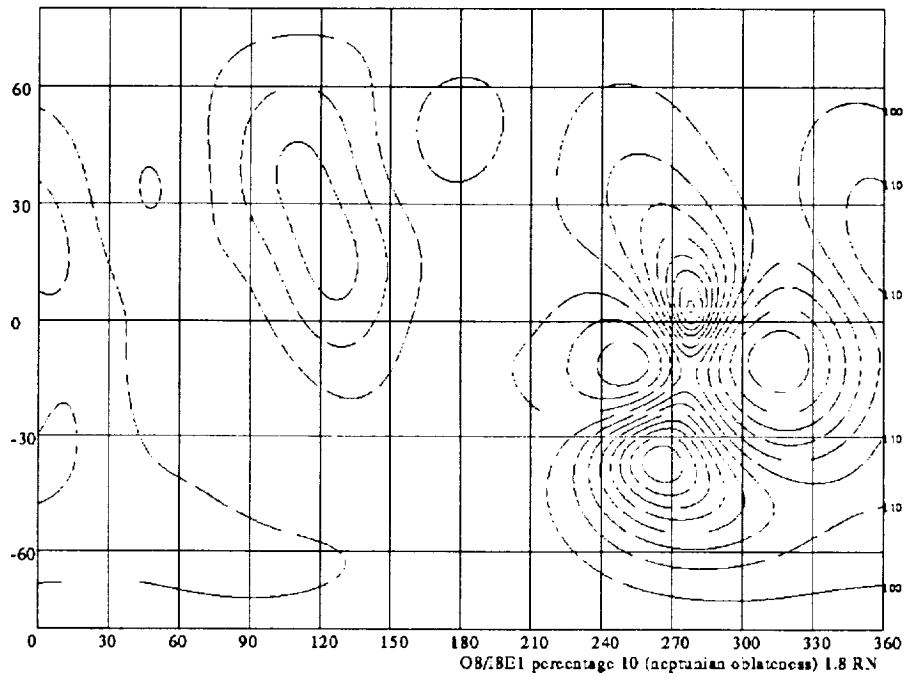


Figure 14

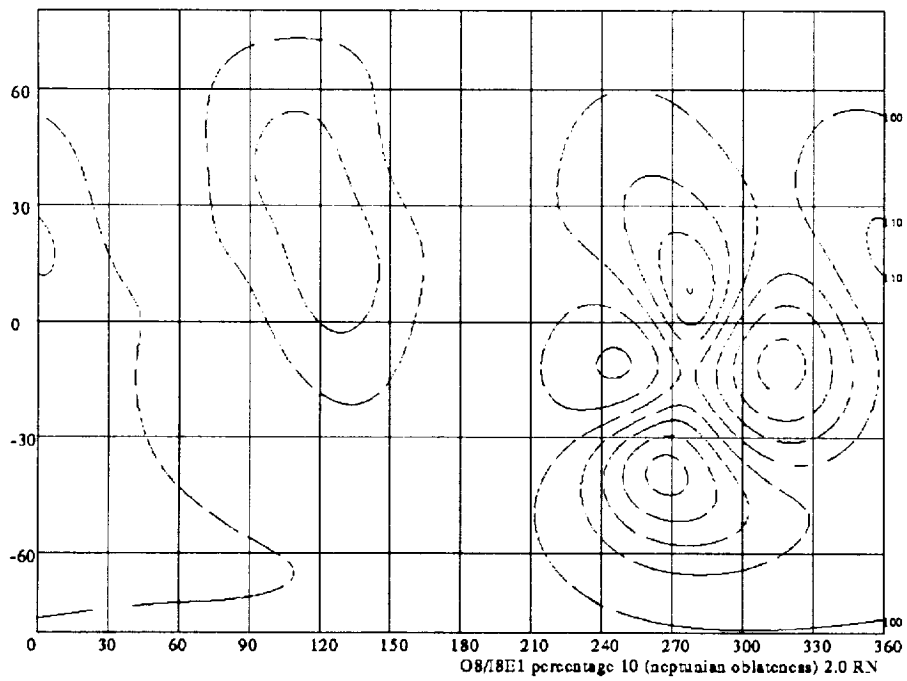


Figure 15

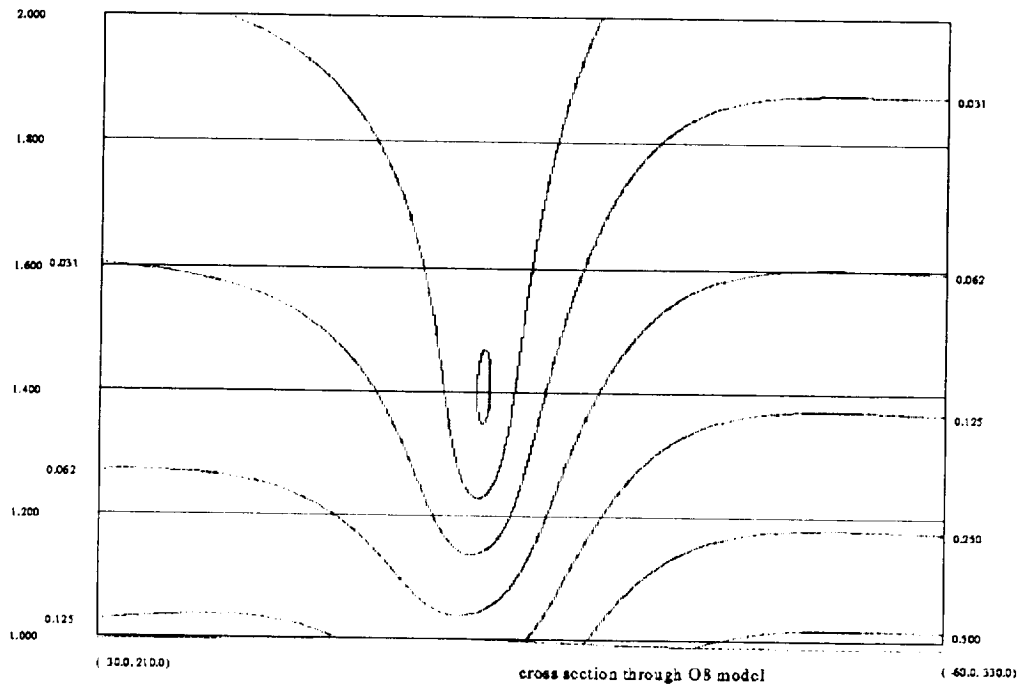


Figure 16

Clearly, then, using the O_8 model to model the field is not a promising prospect, no matter how attractive.

One could arbitrarily cut off the I8E1 44ev model at some order higher than three but lower than eight, thus rendering the complexity of the ensuing model somewhere between the two extremes. However, there seems no rational way in which to decide where to place such a cut-off; further, given the obvious constraints in even the I8E1 44ev model, any lessening of the number of coefficients will simply lead to an even less useful model.

For several reasons, then, the procedure that worked so well at Uranus cannot be contemplated at Neptune: the field is so complicated that calculations with workstation class computers simply take too long to follow particle motions any distance; the complexity of the field leads to a large number of anomalous regions in the planetary magnetosphere; the discrepancy between the relatively accurate model in the northern hemisphere and (almost certainly) inaccurate model in the southern hemisphere means that trapping regions far from the path would likely be artifacts.

Yet another difficulty exists. Some thirteen eigenvectors are omitted from the I8E1 44ev model, on the grounds that they are not necessary to fit the observed data to within the instrumental accuracy. However, linear combinations of these, subject to constraints detailed by Connerney *et al.* (1991), may be added to the I8E1 44ev model without affecting the model field along the spacecraft trajectory. They note that poorly resolved and unresolved components of the model may

be substantially altered by such combinations. Obviously, changing the model so dramatically, will greatly change the possible trapping regions .

Despite these difficulties, an attempt has been made to uncover trapping regions near to the spacecraft path. However, with the computing facilities available, a relatively coarse surface grid had to be adopted, with a typical grid spacing of some 20° , and no such trapping regions have been discovered.

We therefore are forced to conclude that, for the present, a detailed examination of the neptunian magnetosphere for secondary trapping regions is not possible, although it would become so with increased computing power available.

LISTING I QUAD20.CC

Program to calculate the motion of a low energy electron in a general planetary magnetic field

```
1  // C++ code Copyright (C) David R. Evans G4AMJ/NQOI
2
3  // program to trace motion of charged particles in planetary magnetic
4  // fields
5
6  #include <defines.h>
7  #include <magfield.h>
8  #include <vector.h>
9
10 #include <math.h>
11 #include <iostream.h>
12 #include <stdlib.h>
13 #include <sys/time.h>
14 #include <sys/resource.h>
15
16 // here define either URANUS or NEPTUNE
17 #undef URANUS
18 #define NEPTUNE
19
20 // if ORDER3 is defined, then uses 08 instead of full I8E1 44ev
21 #define ORDER3
22
23 #ifdef URANUS
24 const float ellipticity = 0.0229;
25 #else
26 const float ellipticity = 0.017;
27 #endif
28
29 const int LIMIT = 4096;
30
31 magfield* field;
32
33 // define matherr to abort
34 int matherr(struct exception *x)
35 { cout << "MATH ERROR\n";
36   exit(1);
37 }
38
39 // I calculate the following function: (latitude in degrees)
40 // from ellipticity = (re - rp) / rp.
41
42 float radius(const float co_latitude, const float equatorial_radius = 1)
43 { //const float lat_in_radians = latitude * PI / 180,
44   const float lat_in_radians = PI / 2 - co_latitude,
45             C = cos(lat_in_radians),
46             S = sin(lat_in_radians);
```

```

47
48     return sqrt(SQR(equatorial_radius) /
49                  (C * C + S * S * SQR(1 + ellipticity)));
50 }
51
52 extern const cartesian uranus_q3(const spherical&);
53
54 const double    twopi = 2 * PI,
55                field_factor = 1.1,
56                circle_size = 0.01,
57                max_intermediate_points = 10,
58                min_distance_from_previous = 0.01,
59                default_horiz_epsilon = 0.04,
60                e = 1.602e-19,
61                m = 9.109558e-31,
62                c = 2.997925e8,
63                v_cg_factor = 256.0,
64                cycles_per_time_step = 1024.0,
65                min_distance_from_previous_output = 0.1;
66
67 const int       skip = 100,
68                up = 1,
69                down = -up;
70
71 double  q = 1.602e-19,                                // ATT does not permit "q = e"
72        total_time = 0, rotations_so_far = 0;
73
74 float target_field, target_frequency;
75
76 double pitch_angle, alpha, energy;
77
78 // move a distance along a field line
79 cartesian along_field_line(const cartesian& p, const int direction,
80                            const double distance)
81 { const cartesian p1 = p + direction * distance * unit(field->field(p));
82   const cartesian pt = p1 - direction * distance * unit(field->field(p1));
83   const cartesian delta_p = pt - p1;
84   return p1 - 0.5 * delta_p;
85 }
86
87 double radius(const cartesian& p)
88 { const double scale = 0.001;
89   const cartesian p1 = along_field_line(p, up, scale);
90   const cartesian p2 = along_field_line(p, down, scale);
91   const double t1 = length(p1 - p2);
92   const double t2 = angle(p1 - p, p - p2);
93   // const double temp = length(p1 - p2) / (2 * angle(p1 - p, p - p2));
94   return length(p1 - p2) / angle(p1 - p, p - p2);
95 }
96
97 // find a mirror equator point
98 cartesian mirror_equator(const cartesian& init)
99 { const double max_distance = 1000, max_error = 0.001;
100   double delta = 0.01;

```

```

101     int direction = 1;
102     cartesian x_old = init, x_new;
103     cartesian old_field = field->field(init);
104     x_new = along_field_line(x_old, direction, delta);
105     cartesian new_field = field->field(x_new);
106
107     // start off in the correct direction -- i.e. lower B
108     if (length(new_field) > length(old_field)) then direction = -1;
109     while (delta > max_error)
110     { // cout << spherical(x_old) << field->field(x_old) << "\n";
111       x_new = along_field_line(x_old, direction, delta);
112       new_field = field->field(x_new);
113     }
114     // too far from planet?
115     if (length(x_new) > max_distance) then return cartesian(0, 0, 0);
116
117     if (length(new_field) > length(old_field)) then
118     { if (delta > max_error) then
119       { direction = - direction;
120         delta /= 2;
121       }
122     }
123     x_old = x_new;
124     old_field = new_field;
125   }
126   return x_old;
127 }
128
129 cartesian correct_mirror_equator(const cartesian& x_init)
130 { const double max_frac_error = 0.2,
131   max_field_error = 0.001;
132
133   cartesian x0 = x_init;
134   boolean close_enough;
135   do
136   { x0 = mirror_equator(x0);
137     cartesian field0 = field->field(x0);
138     if (field0 == 0) then return cartesian(0, 0, 0);
139   }
140   // walk to a region in which the B is closer to the target
141   double fractional_field_error = (length(field0) - target_field) /
142     length(field0);
143   if (fabs(fractional_field_error) > max_frac_error) then
144     fractional_field_error = SIGN(fractional_field_error) * max_frac_error;
145
146   // are we there yet?
147   close_enough = (fabs(fractional_field_error) <=
148     max_field_error);
149
150   if (!close_enough) then
151   { double new_radius = 1 + fractional_field_error / 10;
152     cartesian temp = x0;
153     x0 *= new_radius;
154

```



```

155 // check for area in which field decreases inwards or increases outwards
156     if ((SIGN(new_radius - 1)) !=
157         (SIGN(length(field0) / length(field->field(x0)) - 1))) then
158     { new_radius = 1 / new_radius;
159       new_radius *= new_radius;
160       x0 *= new_radius;
161       if ((SIGN(new_radius - 1)) ==
162           (SIGN(length(field0) / length(field->field(x0)) - 1))) then
163       { cout << "Field shape problem\n";
164         x0 = temp * (1 + (SIGN(fractional_field_error) * 0.5));
165       }
166     }
167   }
168 } while (!close_enough);
169 return x0;
170 }
171
172 // calculate parallel and perpendicular components of a field
173 void calculate_field_gradient(const cartesian& posn, cartesian& par, cartesian&
per)
174 { const double grid_spacing = 0.001;
175   cartesian b[6];
176   b[0] = field->field(cartesian(x(posn) - grid_spacing / 2, y(posn),
177                                 z(posn)));
178   b[1] = field->field(cartesian(x(posn) + grid_spacing / 2, y(posn),
179                                 z(posn)));
180   b[2] = field->field(cartesian(x(posn), y(posn) - grid_spacing / 2,
181                                 z(posn)));
182   b[3] = field->field(cartesian(x(posn), y(posn) + grid_spacing / 2,
183                                 z(posn)));
184   b[4] = field->field(cartesian(x(posn), y(posn),
185                                 z(posn) - grid_spacing / 2));
186   b[5] = field->field(cartesian(x(posn), y(posn),
187                                 z(posn) + grid_spacing / 2));
188   cartesian del = cartesian(length(b[1]) - length(b[0]),
189                             length(b[3]) - length(b[2]),
190                             length(b[5]) - length(b[4]));
191   del /= grid_spacing;
192   const cartesian& b_field = unit(field->field(posn));
193   par = _dot(b_field, del) * b_field;
194   per = del - par;
195 }
196
197 int do_main(const double in_theta, const double in_phi)
198 { spherical s(1.0, in_theta, in_phi); // the starting position
199
200   cartesian output_posn;
201   int n_out = 0, n_calculated = 0;
202
203   // type_of_run is used to determine how often to produce output.
204   // 'P' means a position-based decision; 'T' indicates a time-based one.
205   const char type_of_run = 'T';
206
207   // convert

```

```

208     energy = 1000;                      // a 1 eV particle
209     alpha = pitch_angle * PI / 180;      // radians
210
211 // we actually start from the mirror equator for the starting field line
212     cartesian posn = mirror_equator(cartesian(s));
213
214     target_field = length(field->field(posn));
215     target_frequency = target_field * 2.8e10;
216
217     const float seconds_per_step = cycles_per_time_step / target_frequency;
218     const int original_theta = (int)(t(spherical(posn)) * 180 / PI);
219     const int end_run_phi = (int)(p(spherical(posn)) * 180 / PI + 1) % 360;
220     const float magnetic_moment = length(field->field(posn)) / SQR(sin(alpha));
221     const float speed = sqrt(2 * e * energy / m) / (25000.0 * 1000);
222
223     cartesian posn_field = field->field(posn);
224     cartesian unit_field = unit(posn_field);
225     boolean finished;
226
227 // now let the electron go
228     do
229     { cartesian grad_par, grad_per;
230       calculate_field_gradient(posn, grad_par, grad_per);
231
232 // calculate v_par -- assume conservative field
233       cartesian v_par = unit_field * speed * cos(alpha);
234
235 // calculate v_cg
236       cartesian v_cg = unit_field * grad_per;
237       double factor = m * SQR(speed) *
238                     (1 + SQR(cos(alpha))) / (2 * e * SQR(length(posn_field)));
239       v_cg *= (factor * v_cg_factor);
240
241 // f_par
242       factor = - (SQR(speed * sin(alpha))) / (2 * length(posn_field));
243       cartesian f_par = factor * grad_par;
244
245 // move the electron under constant acceleration
246       cartesian old_posn = posn;
247
248 // guess at new position
249       cartesian guess_posn = posn + ((v_cg + v_par) * seconds_per_step +
250                                     0.5 * (f_par) * SQR(seconds_per_step));
251       cartesian guess_field = field->field(guess_posn);
252       cartesian guess_unit_field = unit(guess_field);
253 // note that grad_par and grad_per will reference the GUESSED position
254       calculate_field_gradient(guess_posn, grad_par, grad_per);
255       double guess_alpha = (length(guess_field) >= magnetic_moment ? PI / 2 :
256                             asin(sqrt(length(guess_field) / magnetic_moment)));
257       int v_dirn = SIGN(_dot((v_par + f_par * seconds_per_step), guess_field));
258       if (v_dirn < 0) then
259         guess_alpha = PI - guess_alpha;
260       double guess_factor = - (SQR(speed * sin(guess_alpha))) /
261                             (2 * length(guess_field));

```

```

262     cartesian guess_f_par = guess_factor * grad_par;
263
264 // take the mean
265     f_par = (f_par + guess_f_par) / 2;
266
267 // place v_par not quite in the parallel direction (the "field line" bends!!)
268     cartesian temp_v_par = unit(unit_field + guess_unit_field) * length(v_par);
269     v_par = ((_dot(unit_field, v_par) < 0) ? -temp_v_par : temp_v_par);
270
271 // now do it for the drift as well
272     cartesian guess_v_cg = guess_unit_field * grad_per;
273     guess_factor = m * SQR(speed) *
274         (1 + SQR(cos(guess_alpha))) /
275         (2 * e * SQR(length(guess_field)));
276     guess_v_cg *= (guess_factor * v_cg_factor);
277
278 // take the mean
279     v_cg = (v_cg + guess_v_cg) / 2;
280
281     posn += ((v_cg + v_par) * seconds_per_step +
282         0.5 * (f_par) * SQR(seconds_per_step));
283     posn_field = field->field(posn);
284     unit_field = unit(posn_field);
285     if (length(posn - old_posn) > 0.1) then
286     { cout << "*** BIG MOVEMENT\n" << old_posn << posn << (v_cg + v_par) <<
287         (v_cg + v_par) * seconds_per_step <<
288         old_posn + (v_cg + v_par) * seconds_per_step << "\n";
289     }
290
291 // add fpar to give direction wrt B
292     alpha = (length(posn_field) >= magnetic_moment ? PI / 2 :
293         asin(sqrt(length(posn_field) / magnetic_moment)));
294     v_dirn = SIGN(_dot((v_par + f_par * seconds_per_step), posn_field));
295     if (v_dirn < 0) then
296         alpha = PI - alpha;
297
298     total_time += seconds_per_step;
299     double distance_from_previous_output = length(posn - output_posn);
300     n_calculated++;
301
302 // stop if we hit the planet
303     if (r(spherical(posn)) < radius(t(spherical(posn)))) then
304         return (-1);
305
306 // stop if we go too far from the planet
307     if (r(spherical(posn)) > 3.0) then
308         return (-2);
309
310 // possibly print out a description of where we are
311     switch (type_of_run)
312     { case 'P' : case 'p' :
313         if (distance_from_previous_output >= min_distance_from_previous_output)
314         { cout << alpha * 180 / PI << " " << r(spherical(posn)) << " " <<

```

```

315         t(spherical(posn)) * 180 / PI << " " <<
316         p(spherical(posn)) * 180 / PI << "\n";
317     output_posn = posn;
318     n_out++;
319 }
320 break;
321 case 'T' : case 't' :
322     if (!(n_calculated % 100)) then
323         { if (!(n_calculated % 1000)) then
324             cout << alpha * 180 / PI << " " << r(spherical(posn)) << " " <<
325                 t(spherical(posn)) * 180 / PI << " " <<
326                 p(spherical(posn)) * 180 / PI << "\n";
327             n_out++;
328         }
329     break;
330 }
331
332 finished = (n_out > 1000);
333 if (type_of_run == 'T' || type_of_run == 't') then
334     finished = (n_out > LIMIT);
335     spherical s_posn(posn);
336     finished = finished || ((n_out > LIMIT) && ((int)(p(s_posn) * 180 / PI) ==
end_run_phi));
337     finished = finished || ((n_out > LIMIT) && ((int)(t(s_posn) * 180 / PI) ==
original_theta));
338 } while (!finished);
339 // spherical s_posn(posn);
340 cout << "Finished, n_out = " << n_out << "\n\n";
341 //     "\nPhi = " << (int)(p(s_posn) * 180 / PI) << "end phi = " <<
342 //     end_run_phi << "\nTheta = " << (int)(t(s_posn) * 180 / PI) <<
343 //     "end theta = " << original_theta << "\n";
344 return 1;
345 }
346
347 main()
348 { cout << "LIMIT = " << LIMIT << " cycles_per_time_step = " <<
349     cycles_per_time_step << "\n\n";
350
351 // build the correct field model
352 #ifdef URANUS
353     const int order = 2,
354             rank = 2;
355
356 // how many "g"s and "h"s are there?
357     int n_coeffs = 0;
358     for (int n = 1; n <= order; n_coeffs += ++n) ;
359
360     int32 * g, * h;
361
362     heap_check(g = new int32 [n_coeffs]);
363     heap_check(h = new int32 [n_coeffs]);
364
365 // specify the model coefficients
366     h[0] = 0; h[2] = 0;

```

```

367
368     g[0] = 11893;
369     g[1] = 11579;
370     h[1] = -15684;
371     g[2] = -6030;
372     g[3] = -12587;
373     g[4] = 196;
374     h[3] = 6116;
375     h[4] = 4759;
376
377     heap_check(field = new magfield(order, g, h));
378 #endif
379
380 #ifndef NEPTUNE
381     const int order = 8,
382            rank = 8;
383
384     // how many "g"s and "h"s are there?
385     int n_coeffs = 0;
386     for (int n = 1; n <= order; n_coeffs += ++n) ;
387
388     int32 * g, * h;
389
390     heap_check(g = new int32 [n_coeffs]);
391     heap_check(h = new int32 [n_coeffs]);
392
393     // specify the model coefficients
394     h[0] = 0; h[2] = 0; h[5] = 0; h[9] = 0;
395     h[14] = 0; h[20] = 0; h[27] = 0; h[35] = 0;
396
397     // n = 1
398     g[0] = 9732;
399     g[1] = 3220;                h[1] = -9889;
400
401     // n = 2
402     g[2] = 7448;
403     g[3] = 664;                h[3] = 11230;
404     g[4] = 4499;                h[4] = -70;
405
406
407     // n = 3
408     g[5] = -6592;
409     g[6] = 4098;                h[6] = -3669;
410     g[7] = -3581;                h[7] = 1791;
411     g[8] = 484;                h[8] = -770;
412
413     // n = 4
414     g[9] = 2243;
415     g[10] = 557;                h[10] = -1889;
416     g[11] = 3099;                h[11] = 2607;
417     g[12] = -1287;                h[12] = 1204;
418     g[13] = -5073;                h[13] = -456;
419
420     // n = 5

```

```

421     g[14] = -202;
422     g[15] = -229;
423     g[16] = 526;
424     g[17] = -2846;
425     g[18] = -1425;
426     g[19] = -2835;
427
428     // n = 6
429     g[20] = -2175;
430     g[21] = -466;
431     g[22] = -1269;
432     g[23] = -2233;
433     g[24] = -887;
434     g[25] = -496;
435     g[26] = 755;
436
437     // n = 7
438     g[27] = 1671;
439     g[28] = 1678;
440     g[29] = 1625;
441     g[30] = 2157;
442     g[31] = -483;
443     g[32] = 1873;
444     g[33] = 584;
445     g[34] = 664;
446
447     // n = 8
448     g[35] = -689;
449     g[36] = 238;
450     g[37] = -90;
451     g[38] = -1304;
452     g[39] = 311;
453     g[40] = -367;
454     g[41] = -249;
455     g[42] = 1333;
456     g[43] = -1239;
457
458     heap_check(field = new magfield(order, g, h));
459
460     // truncate if 08
461     #ifdef ORDER3
462         field->order(3);
463     #endif
464
465     #endif
466
467
468     do
469     { int dtheta, dphi;
470       cin >> dtheta >> dphi >> pitch_angle;
471       if (!cin.rdstate())
472         cout << "\n" << dtheta << " " << dphi << " " << pitch_angle << "\n";
473       cout << do_main(dtheta * PI / 180, dphi * PI / 180) << "\n\n";
474     } while (!cin.rdstate());

```

```
475     cout << "target frequency = " << target_frequency << "\n\n";
476
477 }
```

LISTING II

a) MAGFIELD.H

Header file of class for the calculation of magnetic fields up to order 8

```

1 // C++ code Copyright (C) David R. Evans G4AMJ/NQOI
2
3 // a general purpose magnetic field class.
4
5 // the real trick here is to make it both general and *fast*.
6
7 // only const magfield should exist
8
9 #include <defines.h>
10 #include <vector.h>
11
12 #include <math.h>
13
14 #define real float
15
16 class magfield
17 { int _order, _max_order;
18   int _n_coeffs;
19
20   // the following is messy and ungeneral, but is necessary for speed
21   // (which is the most important requirement, as the field will need
22   // to be calculated _many_ times.)
23
24   int32 g10, g11,
25         g20, g21, g22,
26         g30, g31, g32, g33,
27         g40, g41, g42, g43, g44,
28         g50, g51, g52, g53, g54, g55,
29         g60, g61, g62, g63, g64, g65, g66,
30         g70, g71, g72, g73, g74, g75, g76, g77,
31         g80, g81, g82, g83, g84, g85, g86, g87, g88;
32
33   int32 h10, h11,
34         h20, h21, h22,
35         h30, h31, h32, h33,
36         h40, h41, h42, h43, h44,
37         h50, h51, h52, h53, h54, h55,
38         h60, h61, h62, h63, h64, h65, h66,
39         h70, h71, h72, h73, h74, h75, h76, h77,
40         h80, h81, h82, h83, h84, h85, h86, h87, h88;
41
42   real * _br, * _bt, * _bp;
43
44
45   // there are a whole slew of constants that we want to make static
46
47 public:
48   static /* const */ real sqrt3,

```


49	sqrt5,
50	sqrt6,
51	sqrt7,
52	sqrt8,
53	sqrt10,
54	sqrt15,
55	sqrt21,
56	sqrt27,
57	sqrt32,
58	sqrt35,
59	sqrt55,
60	sqrt77,
61	sqrt105,
62	sqrt125,
63	sqrt128,
64	sqrt135,
65	sqrt189,
66	sqrt231,
67	sqrt280,
68	sqrt343,
69	sqrt429,
70	sqrt512,
71	sqrt715,
72	sqrt840,
73	sqrt875,
74	sqrt945,
75	sqrt1001,
76	sqrt1029,
77	sqrt1120,
78	sqrt1155,
79	sqrt1512,
80	sqrt2048,
81	sqrt2079,
82	sqrt3003,
83	sqrt3773,
84	sqrt3861,
85	sqrt5145,
86	sqrt8192,
87	sqrt9317,
88	sqrt9856,
89	sqrt10395,
90	sqrt11319,
91	sqrt18711,
92	sqrt30240,
93	sqrt32768,
94	sqrt34749,
95	sqrt86515,
96	sqrt93555,
97	sqrt131072,
98	sqrt446875,
99	sqrt4084101,
100	sqrt4204629,
101	sqrt5872581,
102	sqrt124540416;

```

103
104 public:
105
106 // a magfield is constructed by one of the following mechanisms:
107
108 // 1.
109 // magfield(order, all the "g"s, then all the "h"s, which should be
110 // int32s
111
112     magfield(const int order, const int32, ...);
113
114 // 2. magfield(order, array of all the "g"s, array of all the "h"s
115
116     magfield(const int order, const int32 * const g,
117               const int32 * const h);
118
119 // 3. magfield(order, array of "g"s followed by "h"s)
120
121     magfield(const int order, const int32 * const gh);
122
123 // destructor
124     ~magfield(void);
125
126     cartesian field(const spherical&) const;
127
128 // change the order to which the field is calculated
129     inline int order(void)
130     { return _order; }
131     void order(const int n);
132 };

```

LISTING II
b) MAGFIELD.CC

Class for the calculation of planetary magnetic fields up to order 8

```

1 // C++ code Copyright (C) David R. Evans G4AMJ/NQOI
2
3 // a general purpose magnetic field class
4
5 #include <magfield.h>
6
7 #include <stdarg.h>
8
9 // constructors
10 magfield::magfield(const int order, const int32, ...) : _order(order)
11 { _max_order = _order;
12
13     va_list ap;
14     va_start(ap, order);
15
16     // how many "g"s and "h"s, are there?
17     _n_coeffs = 0;
18     for (int n = 1; n <= _order; _n_coeffs += ++n) ;
19
20     // allocate space for the components
21     _br = new real [_order + 1];
22     _bt = new real [_order + 1];
23     _bp = new real [_order + 1];
24
25     int32 * _g, * _h;
26
27     // allocate the space for the coefficients
28     heap_check(_g = new int32 [_n_coeffs]);
29     heap_check(_h = new int32 [_n_coeffs]);
30
31     // copy the coefficients from the parameter list to the internal arrays
32     for (n = 0; n < _n_coeffs; n++)
33         _g[n] = va_arg(ap, const int32);
34     for (n = 0; n < _n_coeffs; n++)
35         _h[n] = va_arg(ap, const int32);
36
37     va_end(ap);
38
39     // now build an easier way of referring to the arrays; we do this solely
40     // for speed later
41
42     int32 ** gp, ** hp;
43     heap_check(gp = new int32* [_order + 1]); // so the "gp"s go wrt 1
44     heap_check(hp = new int32* [_order + 1]);
45
46     int index = 0;
47     for (n = 1; n <= _order; n++)
48     { gp[n] = &(_g[index]);

```

```

49     hp[n] = &(_h[index]);
50     index += (n + 1);
51 }
52
53 // the following is incredibly unsubtle, but I can't think of a more
54 // elegant way to do it
55
56 if (_order > 0) then
57 { g10 = gp[1][0];
58   g11 = gp[1][1];
59   h10 = hp[1][0];
60   h11 = hp[1][1];
61 }
62
63 if (_order > 1) then
64 { g20 = gp[2][0];
65   g21 = gp[2][1];
66   g22 = gp[2][2];
67   h20 = hp[2][0];
68   h21 = hp[2][1];
69   h22 = hp[2][2];
70 }
71
72 if (_order > 2) then
73 { g30 = gp[3][0];
74   g31 = gp[3][1];
75   g32 = gp[3][2];
76   g33 = gp[3][3];
77   h30 = hp[3][0];
78   h31 = hp[3][1];
79   h32 = hp[3][2];
80   h33 = hp[3][3];
81 }
82
83 if (_order > 3) then
84 { g40 = gp[4][0];
85   g41 = gp[4][1];
86   g42 = gp[4][2];
87   g43 = gp[4][3];
88   g44 = gp[4][4];
89   h40 = hp[4][0];
90   h41 = hp[4][1];
91   h42 = hp[4][2];
92   h43 = hp[4][3];
93   h44 = hp[4][4];
94 }
95
96 if (_order > 4) then
97 { g50 = gp[5][0];
98   g51 = gp[5][1];
99   g52 = gp[5][2];
100  g53 = gp[5][3];
101  g54 = gp[5][4];
102  g55 = gp[5][5];

```

```

103     h50 = hp[5][0];
104     h51 = hp[5][1];
105     h52 = hp[5][2];
106     h53 = hp[5][3];
107     h54 = hp[5][4];
108     h55 = hp[5][5];
109 }
110
111 if (_order > 5) then
112 { g60 = gp[6][0];
113   g61 = gp[6][1];
114   g62 = gp[6][2];
115   g63 = gp[6][3];
116   g64 = gp[6][4];
117   g65 = gp[6][5];
118   g66 = gp[6][6];
119   h60 = hp[6][0];
120   h61 = hp[6][1];
121   h62 = hp[6][2];
122   h63 = hp[6][3];
123   h64 = hp[6][4];
124   h65 = hp[6][5];
125   h66 = hp[6][6];
126 }
127
128 if (_order > 6) then
129 { g70 = gp[7][0];
130   g71 = gp[7][1];
131   g72 = gp[7][2];
132   g73 = gp[7][3];
133   g74 = gp[7][4];
134   g75 = gp[7][5];
135   g76 = gp[7][6];
136   g77 = gp[7][7];
137   h70 = hp[7][0];
138   h71 = hp[7][1];
139   h72 = hp[7][2];
140   h73 = hp[7][3];
141   h74 = hp[7][4];
142   h75 = hp[7][5];
143   h76 = hp[7][6];
144   h77 = hp[7][7];
145 }
146
147 if (_order > 7) then
148 { g80 = gp[8][0];
149   g81 = gp[8][1];
150   g82 = gp[8][2];
151   g83 = gp[8][3];
152   g84 = gp[8][4];
153   g85 = gp[8][5];
154   g86 = gp[8][6];
155   g87 = gp[8][7];
156   g88 = gp[8][8];

```

```

157     h80 = hp[8][0];
158     h81 = hp[8][1];
159     h82 = hp[8][2];
160     h83 = hp[8][3];
161     h84 = hp[8][4];
162     h85 = hp[8][5];
163     h86 = hp[8][6];
164     h87 = hp[8][7];
165     h88 = hp[8][8];
166 }
167
168 destroy_array(gp);
169 destroy_array(hp);
170 destroy_array(_g);
171 destroy_array(_h);
172 }
173
174 magfield::magfield(const int order, const int32 * const g,
175                   const int32 * const h) : _order(order)
176 { _max_order = _order;
177
178 // how many "g"s and "h"s, are there?
179 _n_coeffs = 0;
180 for (int n = 1; n <= _order; _n_coeffs += ++n) ;
181
182 // allocate space for the components
183 _br = new real [_order + 1];
184 _bt = new real [_order + 1];
185 _bp = new real [_order + 1];
186
187 // now build an easier way of referring to the arrays; we do this solely
188 // for speed later
189
190 int32 ** gp, ** hp;
191 heap_check(gp = new int32* [_order + 1]); // so the "gp"s go wrt 1
192 heap_check(hp = new int32* [_order + 1]);
193
194 int index = 0;;
195 for (n = 1; n <= _order; n++)
196 { gp[n] = &(g[index]);
197   hp[n] = &(h[index]);
198   index += (n + 1);
199 }
200
201 // the following is incredibly unsubtle, but I can't think of a more
202 // elegant way to do it
203
204 if (_order > 0) then
205 { g10 = gp[1][0];
206   g11 = gp[1][1];
207   h10 = hp[1][0];
208   h11 = hp[1][1];
209 }
210

```

```

211     if (_order > 1) then
212     { g20 = gp[2][0];
213       g21 = gp[2][1];
214       g22 = gp[2][2];
215       h20 = hp[2][0];
216       h21 = hp[2][1];
217       h22 = hp[2][2];
218     }
219
220     if (_order > 2) then
221     { g30 = gp[3][0];
222       g31 = gp[3][1];
223       g32 = gp[3][2];
224       g33 = gp[3][3];
225       h30 = hp[3][0];
226       h31 = hp[3][1];
227       h32 = hp[3][2];
228       h33 = hp[3][3];
229     }
230
231     if (_order > 3) then
232     { g40 = gp[4][0];
233       g41 = gp[4][1];
234       g42 = gp[4][2];
235       g43 = gp[4][3];
236       g44 = gp[4][4];
237       h40 = hp[4][0];
238       h41 = hp[4][1];
239       h42 = hp[4][2];
240       h43 = hp[4][3];
241       h44 = hp[4][4];
242     }
243
244     if (_order > 4) then
245     { g50 = gp[5][0];
246       g51 = gp[5][1];
247       g52 = gp[5][2];
248       g53 = gp[5][3];
249       g54 = gp[5][4];
250       g55 = gp[5][5];
251       h50 = hp[5][0];
252       h51 = hp[5][1];
253       h52 = hp[5][2];
254       h53 = hp[5][3];
255       h54 = hp[5][4];
256       h55 = hp[5][5];
257     }
258
259     if (_order > 5) then
260     { g60 = gp[6][0];
261       g61 = gp[6][1];
262       g62 = gp[6][2];
263       g63 = gp[6][3];
264       g64 = gp[6][4];

```

```

265     g65 = gp[6][5];
266     g66 = gp[6][6];
267     h60 = hp[6][0];
268     h61 = hp[6][1];
269     h62 = hp[6][2];
270     h63 = hp[6][3];
271     h64 = hp[6][4];
272     h65 = hp[6][5];
273     h66 = hp[6][6];
274 }
275
276 if (_order > 6) then
277 { g70 = gp[7][0];
278   g71 = gp[7][1];
279   g72 = gp[7][2];
280   g73 = gp[7][3];
281   g74 = gp[7][4];
282   g75 = gp[7][5];
283   g76 = gp[7][6];
284   g77 = gp[7][7];
285   h70 = hp[7][0];
286   h71 = hp[7][1];
287   h72 = hp[7][2];
288   h73 = hp[7][3];
289   h74 = hp[7][4];
290   h75 = hp[7][5];
291   h76 = hp[7][6];
292   h77 = hp[7][7];
293 }
294
295 if (_order > 7) then
296 { g80 = gp[8][0];
297   g81 = gp[8][1];
298   g82 = gp[8][2];
299   g83 = gp[8][3];
300   g84 = gp[8][4];
301   g85 = gp[8][5];
302   g86 = gp[8][6];
303   g87 = gp[8][7];
304   g88 = gp[8][8];
305   h80 = hp[8][0];
306   h81 = hp[8][1];
307   h82 = hp[8][2];
308   h83 = hp[8][3];
309   h84 = hp[8][4];
310   h85 = hp[8][5];
311   h86 = hp[8][6];
312   h87 = hp[8][7];
313   h88 = hp[8][8];
314 }
315
316 destroy_array(gp);
317 destroy_array(hp);
318

```



```

319 }
320
321 // destructor
322 magfield::~magfield(void)
323 { destroy_array(_br);
324   destroy_array(_bt);
325   destroy_array(_bp);
326 }
327
328 // change the order to which the field is calculated
329 void magfield::order(const int n)
330 { _order = MIN(n, _max_order);
331 }
332
333 // return the value of the field
334 cartesian magfield::field(const spherical& pos) const
335 { const real r = pos.r(),
336   t = pos.t(),
337   p = pos.p();
338
339 // define some handy macros (very un-C++, this)
340 // GC_PLUS_HS == g * c + h * s
341 #define GC_PLUS_HS(x, y) (g##x##y * cnp##y + h##x##y * snp##y)
342 // GS_MINUS_HC == g * s - h * c
343 #define GS_MINUS_HC(x, y) (g##x##y * snp##y - h##x##y * cnp##y)
344
345 // n = 1
346 if (_order >= 1) then
347 { const real cp = cos(p),
348   sp = sin(p),
349   cnp1 = cp,
350   snp1 = sp,
351   r3 = r * r * r,
352   ct = cos(t),
353   st = sin(t);
354
355   const real gc_plus_hs_11 = GC_PLUS_HS(1,1);
356
357   _br[1] = (2 / r3) * (ct * g10 + st * gc_plus_hs_11);
358   _bt[1] = (g10 * st - ct * gc_plus_hs_11) / r3;
359   _bp[1] = GS_MINUS_HC(1,1) / r3;
360
361 // n = 2
362 if (_order >= 2) then
363 { const real cnp2 = cos(2 * p),
364   snp2 = sin(2 * p),
365   cnt2 = cos(2 * t),
366   snt2 = sin(2 * t),
367   stn2 = st * st,
368   r4 = r3 * r;
369
370   const real gc_plus_hs_22 = GC_PLUS_HS(2,2),
371   gc_plus_hs_21 = GC_PLUS_HS(2,1);
372

```

```

373      _br[2] = 3 * (1 + 3 * cnt2) * g20 / (4 * r4) + sqrt27 * (gc_plus_hs_22) *
374      stn2 / (2 * r4) + sqrt27 * ct * (gc_plus_hs_21) * st / r4;
375
376
377 // bug required changing sign of _bt[2] and removal of factor of st
378 // in the (2,1) term
379      _bt[2] = - ((sqrt3 * cnt2 * (1 / st) * (cp * g21 + h21 * sp) * st / r3 - 3
* g20 *
380      snt2 / (2 * r3) + sqrt3 * (cnp2 * g22 + h22 * snp2) * snt2 /
(2 * r3)) / r);
381
382      _bp[2] = - ((1 / st) * (sqrt3 * (cnp2 * h22 - g22 * snp2) * stn2 / r3 +
sqrt3 * ct
383      * (cp * h21 - g21 * sp) * st / r3) / r);
384
385 // n = 3
386
387      if (_order >= 3) then
388      { const real r5 = r4 * r,
389          cnp3 = cos(3 * p),
390          snp3 = sin(3 * p),
391          cnt3 = cos(3 * t),
392          ctn2 = ct * ct,
393          ctn3 = ctn2 * ct,
394          stn3 = stn2 * st;
395
396          const real V4r5 = 4 * r5,
397          Vsqr8r5 = sqrt8 * r5;
398
399          const real gc_plus_hs_33 = GC_PLUS_HS(3,3),
400          gc_plus_hs_32 = GC_PLUS_HS(3,2),
401          gc_plus_hs_31 = GC_PLUS_HS(3,1);
402
403          _br[3] = (3 * ct + 5 * cnt3) * g30 / (2 * r5) + 2 * sqrt15 * ct * (cnp2
* g32 + h32
404      * snp2) * stn2 / r5 - sqrt6 * (1 - 5 * ctn2) * (cp * g31 + h31 * sp) * st
405      / r5 + sqrt10 * (cnp3 * g33 + h33 * snp3) * stn3 / r5;
406
407
408 // Simplify[bt2[3]]
409      _bt[3] = (3 * (3 + 5 * cnt2) * g30 * st / (4 * r4) - sqrt15 * (1 + 3 * cnt2) *
410      (cnp2 * g32 + h32 * snp2) * st / (4 * r4) - sqrt3 / sqrt128 * (ct + 15 *
411      cnt3) * (1 / st) * (cp * g31 + h31 * sp) * st / r4 - 3 * sqrt5 / sqrt8 *
412      (ct / st) * (cnp3 * g33 + h33 * snp3) * stn3 / r4) / r;
413
414 // BP[3]
415      _bp[3] = - ((1 / st) * (sqrt15 * ct * (cnp2 * h32 - g32 * snp2) * stn2 / r4 +
416      sqrt3 / sqrt32 * (3 + 5 * cnt2) * (cp * h31 - g31 * sp) * st / r4 + 3 *
417      sqrt5 / sqrt8 * (cnp3 * h33 - g33 * snp3) * stn3 / r4) / r);
418
419 // n = 4
420      if (_order >= 4) then
421      { const real r6 = r5 * r,
422          cnp4 = cos(4 * p),

```

```

423             snp4 = sin(4 * p),
424             cnt4 = cos(4 * t),
425             ctn3 = ctn2 * ct,
426             ctn4 = ctn3 * ct,
427             stn4 = stn3 * st;
428
429             const real Vsqr8r6 = sqrt8 * r6,
430             V4r6 = 4 * r6,
431             V8r6 = 8 * r6,
432             Vstr6 = st * r6;
433
434             const real gc_plus_hs_44 = GC_PLUS_HS(4,4),
435             gc_plus_hs_43 = GC_PLUS_HS(4,3),
436             gc_plus_hs_42 = GC_PLUS_HS(4,2),
437             gc_plus_hs_41 = GC_PLUS_HS(4,1);
438
439             _br[4] = 5 * (3 - 30 * ctn2 + 35 * ctn4) * g40 / (8 * r6) + sqrt125 * (5 + 7 *
440 cnt2) * (cnp2 * g42 + h42 * snp2) * stn2 / (8 * r6) + sqrt875 * (cnp4 *
441 g44 + h44 * snp4) * stn4 / (8 * r6) - sqrt125 / sqrt8 * (3 * ct - 7 *
442 ctn3) * (cp * g41 + h41 * sp) * st / r6 + sqrt875 / sqrt8 * ct * (cnp3 *
443 g43 + h43 * snp3) * stn3 / r6;
444
445             _bt[4] = (5 * (9 * ct + 7 * cnt3) * g40 * st / (8 * r5) - sqrt5 * (5 * ct + 7
*
446 cnt3) * (cnp2 * g42 + h42 * snp2) * st / (4 * r5) - sqrt35 * ct * (cnp4 *
447 g44 + h44 * snp4) * stn3 / (2 * r5) - sqrt5 / sqrt32 * (cnt2 + 7 * cnt4) *
448 (1 / st) * (cp * g41 + h41 * sp) * st / r5 - sqrt35 / sqrt8 * (1 + 2 *
449 cnt2) * (1 / st) * (cnp3 * g43 + h43 * snp3) * stn3 / r5) / r;
450
451             _bp[4] = - ((1 / st) * (sqrt5 * (5 + 7 * cnt2) * (cnp2 * h42 - g42 * snp2) *
stn2
452 / (4 * r5) + sqrt35 * (cnp4 * h44 - g44 * snp4) * stn4 / (2 * r5) - sqrt5
453 / sqrt8 * (3 * ct - 7 * ctn3) * (cp * h41 - g41 * sp) * st / r5 + 3 *
454 sqrt35 / sqrt8 * ct * (cnp3 * h43 - g43 * snp3) *
stn3 / r5) / r);
455
456             // n = 5
457
458             if (_order >= 5) then
459             { const real r7 = r6 * r,
460             ctn5 = ctn4 * ct,
461             stn5 = stn4 * st,
462             cnt5 = cos(5 * t),
463             cnp5 = cos(5 * p),
464             snp5 = sin(5 * p);
465
466             const real gc_plus_hs_55 = GC_PLUS_HS(5,5),
467             gc_plus_hs_54 = GC_PLUS_HS(5,4),
468             gc_plus_hs_53 = GC_PLUS_HS(5,3),
469             gc_plus_hs_52 = GC_PLUS_HS(5,2),
470             gc_plus_hs_51 = GC_PLUS_HS(5,1);
471
472             const real V2r7 = 2 * r7,
473             V4r7 = 4 * r7,

```

```

474      Vsqr32r7 = sqrt32 * r7;
475
476      _br[5] = 3 * (15 * ct - 70 * ctn3 + 63 * ctn5) * g50 / (4 * r7) + sqrt945 * (5
* ct
477 + 3 * cnt3) * (cnp2 * g52 + h52 * snp2) * stn2 / (8 * r7) + 9 * sqrt35 *
478 ct * (cnp4 * g54 + h54 * snp4) * stn4 / (4 * r7) - sqrt135 * (- 1 + 14 *
479 ctn2 - 21 * ctn4) * (cp * g51 + h51 * sp) * st / (4 * r7) + 3 * sqrt35 /
480 sqrt32 * (- 1 + 9 * ctn2) * (cnp3 * g53 + h53 * snp3) * stn2 * st / r7 +
481 9 * sqrt7 / sqrt32 * (cnp5 * g55 + h55 * snp5) * stn5 / r7;
482
483      const real Vstr7 = st * r7,
484      Vsqr128r7 = sqrt128 * r7,
485      V8r7 = 8 * r7;
486
487      _bt[5] = (15 * (15 + 28 * cnt2 + 21 * cnt4) * g50 * st / (64 * r6) - sqrt105 *
(5 +
488 12 * cnt2 + 15 * cnt4) * (cnp2 * g52 + h52 * snp2) * st / (32 * r6) - 3 *
489 sqrt35 * (3 + 5 * cnt2) * (cnp4 * g54 + h54 * snp4) * stn3 / (16 * r6) -
490 sqrt15 * (2 * ct + 21 * cnt3 + 105 * cnt5) * (1 / st) * (cp * g51 + h51 *
491 sp) * st / (128 * r6) - 15 * sqrt7 / sqrt128 * (ct / st) * (cnp5 * g55 +
492 h55 * snp5) * stn5 / r6 - 3 * sqrt35 / sqrt2048 * (1 + 15 * cnt2) * (cnp3
493 * g53 + h53 * snp3) * st * stn2 / r6) / r;
494
495      _bp[5] = - ((1 / st) * (sqrt105 * (5 * ct + 3 * cnt3) * (cnp2 * h52 - g52 *
snp2)
496 * stn2 / (8 * r6) + 3 * sqrt35 * ct * (cnp4 * h54 - g54 * snp4) * stn4 /
497 (2 * r6) - sqrt15 * (- 1 + 14 * ctn2 - 21 * ctn4) * (cp * h51 - g51 * sp)
498 * st / (8 * r6) + 3 * sqrt35 / sqrt128 * (- 1 + 9 * ctn2) *
499 (cnp3 * h53 -
500 g53 * snp3) * stn2 * st / r6 + 15 * sqrt7 / sqrt128 * (cnp5 * h55 - g55 *
501 snp5) * stn5 / r6) / r);
502
503      // n = 6
504      if (_order >= 6) then
505      {
506          const real r8 = r7 * r,
507          ctn6 = ctn5 * ct,
508          stn6 = stn5 * st,
509          cnt6 = cos(6 * t),
510          cnp6 = cos(6 * p),
511          snp6 = sin(6 * p);
512
513          const real gc_plus_hs_66 = GC_PLUS_HS(6,6),
514          gc_plus_hs_65 = GC_PLUS_HS(6,5),
515          gc_plus_hs_64 = GC_PLUS_HS(6,4),
516          gc_plus_hs_63 = GC_PLUS_HS(6,3),
517          gc_plus_hs_62 = GC_PLUS_HS(6,2),
518          gc_plus_hs_61 = GC_PLUS_HS(6,1);
519
520      const real Vsqr128r8 = sqrt128 * r8,
521      Vsqr512r8 = sqrt512 * r8,
522      V8r8 = 8 * r8,
523      V16r8 = 16 * r8,
524      V32r8 = 32 * r8,

```

```

525          V128r8 = 128 * r8,
526          Vstr8 = st * r8;
527
528      _br[6] = 7 * ( - 5 + 105 * ctn2 - 315 * ctn4 + 231 * ctn6) * g60 / (16 * r8) +
529      sqrt5145 / sqrt512 * (1 - 19 * ctn2 + 51 * ctn4 - 33 * ctn6) * (cnp2 * g62
530      + h62 * snp2) / r8 + 3 * sqrt343 * (9 + 11 * cnt2) * (cnp4 * g64 + h64 *
531      snp4) * stn4 / (32 * r8) + sqrt11319 / sqrt512 * (cnp6 * g66 + h66 * snp6)
532      * stn6 / r8 - sqrt1029 * ( - 5 * ct + 30 * ctn3 - 33 * ctn5) * (cp * g61 +
533      h61 * sp) * st / (8 * r8) + sqrt5145 / sqrt128 * ( - 3 * ct + 11 * ctn3) *
534      (cnp3 * g63 + h63 * snp3) * stn2 * st / r8 + 3 * sqrt3773 / sqrt128 * ct *
535      (cnp5 * g65 + h65 * snp5) * stn5 / r8;
536
537      _bt[6] = - (( - 21 * (50 * ct + 45 * cnt3 + 33 * cnt5) * g60 * st / (128 *
r7) +
538      sqrt105 / sqrt32768 * (70 * ct + 87 * cnt3 + 99 * cnt5) * (cnp2 * g62 +
539      h62 * snp2) * st / r7 + 3 * sqrt7 * (47 * ct + 33 * cnt3) * (cnp4 * g64 +
540      h64 * snp4) * stn3 / (32 * r7) + sqrt2079 / sqrt128 * ct * (cnp6 * g66 +
541      h66 * snp6) * stn5 / r7 + sqrt21 * (5 * cnt2 + 24 * cnt4 + 99 * cnt6) * (1
542      / st) * (cp * g61 + h61 * sp) * st / (128 * r7) + sqrt945 / sqrt2048 * (7
543      + 14 * cnt2 + 11 * cnt4) * (cnp3 * g63 + h63 * snp3) * st * st / r7 + 3 *
544      sqrt77 / sqrt128 * (2 + 3 * cnt2) * (1 / st) * (cnp5 * g65 + h65 * snp5) *
545      stn5 / r7) / r);
546
547      _bp[6] = - ((1 / st) * (sqrt105 / sqrt128 * (1 - 19 * ctn2 + 51 * ctn4 - 33 *
548      ctn6) * (cnp2 * h62 - g62 * snp2) / r7 + 3 * sqrt7 * (9 + 11 * cnt2) *
549      (cnp4 * h64 - g64 * snp4) * stn4 / (8 * r7) + sqrt2079 / sqrt128 * (cnp6 *
550      h66 - g66 * snp6) * stn6 / r7 - sqrt21 * ( - 5 * ct + 30 * ctn3 - 33 *
551      ctn5) * (cp * h61 - g61 * sp) * st / (8 * r7) + sqrt945 / sqrt128 * ( - 3
552      * ct + 11 * ctn3) * (cnp3 * h63 - g63 * snp3) * stn2 * st / r7 + 15 *
553      sqrt77 / sqrt128 * ct * (cnp5 * h65 - g65 * snp5) * stn5 / r7) / r);
554
555      // n = 7
556      if (_order >= 7) then
557      {const real r9 = r8 * r,
558          ctn7 = ctn6 * ct,
559      ctn8 = ctn7 * ct,
560          stn7 = stn6 * st,
561          cnt7 = cos(7 * t),
562          cnp7 = cos(7 * p),
563          snp7 = sin(7 * p);
564
565          const real gc_plus_hs_77 = GC_PLUS_HS(7,7),
566          gc_plus_hs_76 = GC_PLUS_HS(7,6),
567          gc_plus_hs_75 = GC_PLUS_HS(7,5),
568          gc_plus_hs_74 = GC_PLUS_HS(7,4),
569          gc_plus_hs_73 = GC_PLUS_HS(7,3),
570          gc_plus_hs_72 = GC_PLUS_HS(7,2),
571          gc_plus_hs_71 = GC_PLUS_HS(7,1);
572
573      const real V2r9 = 2 * r9,
574          V4r9 = 4 * r9,
575          V32r9 = 32 * r9,
576          Vsqrt8r9 = sqrt8 * r9,
577          Vstr9 = st * r9,

```

```

578      Vsqr512str9 = sqrt512 * Vstr9;
579
580      _br[7] = ( - 35 * ct + 315 * ctn3 - 693 * ctn5 + 429 * ctn7) * g70 / (2 * r9) +
581      sqrt21 / sqrt2048 * (350 * ct + 275 * cnt3 + 143 * cnt5) * (cnp2 * g72 +
582      h72 * snp2) * stn2 / r9 + sqrt231 * (27 * ct + 13 * cnt3) * (cnp4 * g74 +
583      h74 * snp4) * stn4 / (8 * r9) + sqrt3003 / sqrt8 * ct * (cnp6 * g76 + h76
584      * snp6) * stn6 / r9 - sqrt7 * (5 - 135 * ctn2 + 495 * ctn4 - 429 * ctn6) *
585      (cp * g71 + h71 * sp) * st / (4 * r9) + sqrt21 * (3 - 66 * ctn2 + 143 *
586      ctn4) * (cnp3 * g73 + h73 * snp3) * stn2 * st / (4 * r9) - sqrt231 * (1 -
587      13 * ctn2) * (cnp5 * g75 + h75 * snp5) * stn4 * st / (4 * r9) + sqrt429 *
588      (cnp7 * g77 + h77 * snp7) * stn7 / (4 * r9);
589
590      _bt[7] = - (( - (( - 7 * ( - 5 + 105 * ctn2 - 315 * ctn4 + 231 * ctn6) / 16 +
7 *
591      ct * ( - 35 * ct + 315 * ctn3 - 693 * ctn5 + 429 * ctn7) / 16) * g70 * st
592      / (r8 * ( - stn2))) + ( - 21 * st * ( - 5 * ct + 30 * ctn3 - 33 * ctn5) +
593      49 * ct * st * (5 - 135 * ctn2 + 495 * ctn4 - 429 * ctn6) / 16) * (cp *
594      g71 + h71 * sp) * st / (2 * sqrt7 * r8 * ( - stn2)) - ( - 945 * (1 - 19 *
595      ctn2 + 51 * ctn4 - 33 * ctn6) / 8 + 441 * ct * (15 * ct - 125 * ctn3 + 253
596      * ctn5 - 143 * ctn7) / 8) * (cnp2 * g72 + h72 * snp2) * st / (sqrt1512 *
597      r8 * ( - stn2)) + ( - 1575 * st * ( - stn2) * ( - 3 * ct + 11 * ctn3) +
598      2205 * ct * st * ( - stn2) * (3 - 66 * ctn2 + 143 * ctn4) / 8) * (cnp3 *
599      g73 + h73 * snp3) * st / (20 * sqrt189 * r8 * ( - stn2)) - ( - 10395 * ( -
600      1 + 13 * ctn2 - 23 * ctn4 + 11 * ctn6) / 2 + 24255 * ct * ( - 3 * ct + 19
601      * ctn3 - 29 * ctn5 + 13 * ctn7) / 2) * (cnp4 * g74 + h74 * snp4) * st /
602      (40 * sqrt2079 * r8 * ( - stn2)) + (124740 * ct * stn5 + 72765 * ct * (1 -
603      13 * ctn2) * st * stn4 / 2) * (cnp5 * g75 + h75 * snp5) *
604      st / (80 * sqrt18711 * r8 * ( - stn2)) - ( - 135135 * (1 - 3 * ctn2 + 3 *
605      ctn4 - ctn6) + 945945 * ct * (ct - 3 * ctn3 + 3 * ctn5 - ctn7)) * (cnp6 *
606      g76 + h76 * snp6) * st / (5 * sqrt124540416 * r8 * ( - stn2)) - 7 *
607      sqrt429 * ct * stn7 * (cnp7 * g77 + h77 * snp7) * st / (32 * r8 * ( -
608      stn2))) / r);
609
610      _bp[7] = - ((1 / st) * ( - (sqrt7 * st * (5 - 135 * ctn2 + 495 * ctn4 - 429 *
611      ctn6) * (cp * h71 - g71 * sp)) / (32 * r8) + sqrt21 / sqrt512 * (15 * ct -
612      125 * ctn3 + 253 * ctn5 - 143 * ctn7) * (2 * cnp2 * h72 - 2 * g72 * snp2)
613      / r8 - sqrt21 * st * ( - stn2) * (3 - 66 * ctn2 + 143 * ctn4) * (3 * cnp3
614      * h73 - 3 * g73 * snp3) / (32 * r8) + sqrt231 * ( - 3 * ct + 19 * ctn3 -
615      29 * ctn5 + 13 * ctn7) * (4 * cnp4 * h74 - 4 * g74 * snp4) / (16 * r8) -
616      sqrt231 * (1 - 13 * ctn2) * st * stn4 * (5 * cnp5 * h75 - 5 * g75 * snp5)
617      / (32 * r8) + sqrt3003 / sqrt512 * (ct - 3 * ctn3 + 3 * ctn5 - ctn7) * (6
618      * cnp6 * h76 - 6 * g76 * snp6) / r8 + sqrt429 * stn7 * (7 * cnp7 * h77 - 7
619      * g77 * snp7) / (32 *
r8)) / r);
620
621      if (_order >= 8) then
622      { // n = 8
623
624      const real r10 = r9 * r,
625      ctn9 = ctn8 * ct,
626      stn8 = stn7 * st,
627      cnt8 = cos(6 * t),
628      cnp8 = cos(6 * p),
629      snp8 = sin(6 * p);

```

```

630
631     const real V4r10 = 4 * r10,
632             V32r10 = 32 * r10,
633             V128r10 = 128 * r10,
634             Vsqrt2048r10 = sqrt2048 * r10,
635             Vstr10 = r10 * st,
636             V4str10 = 4 * Vstr10,
637             V8str10 = 2 * V4str10,
638             V16str10 = 2 * V8str10,
639             V32str10 = 2 * V16str10,
640             Vsqrt32str10 = sqrt32 * Vstr10,
641             Vsqrt128str10 = sqrt128 * Vstr10;
642
643     const real Vcpg81 = cp * g81,
644             Vcnp2g82 = cnp2 * g82,
645             Vcnp3g83 = cnp3 * g83,
646             Vcnp4g84 = cnp4 * g84,
647             Vcnp5g85 = cnp5 * g85,
648             Vcnp6g86 = cnp6 * g86,
649             Vcnp7g87 = cnp7 * g87,
650             Vcnp8g88 = cnp8 * g88,
651             Vsph81 = sp * h81,
652             Vsnp2h82 = snp2 * h82,
653             Vsnp3h83 = snp3 * h83,
654             Vsnp4h84 = snp4 * h84,
655             Vsnp5h85 = snp5 * h85,
656             Vsnp6h86 = snp6 * h86,
657             Vsnp7h87 = snp7 * h87,
658             Vsnp8h88 = snp8 * h88;
659
660     const real V3sqrt715 = 3 * sqrt715;
661
662     const real gc_plus_hs_88 = GC_PLUS_HS(8,8),
663             gc_plus_hs_87 = GC_PLUS_HS(8,7),
664             gc_plus_hs_86 = GC_PLUS_HS(8,6),
665             gc_plus_hs_85 = GC_PLUS_HS(8,5),
666             gc_plus_hs_84 = GC_PLUS_HS(8,4),
667             gc_plus_hs_83 = GC_PLUS_HS(8,3),
668             gc_plus_hs_82 = GC_PLUS_HS(8,2),
669             gc_plus_hs_81 = GC_PLUS_HS(8,1);
670
671     _br[8] = 9 * (35 - 1260 * ctn2 + 6930 * ctn4 - 12012 * ctn6 + 6435 * ctn8) *
g80 /
672 (128 * r10) + 27 * sqrt35 / sqrt2048 * (35 / 128 + cnt2 / 2 + 11 * cnt4 /
673 32 - 143 * cnt8 / 128) * (cnp2 * g82 + h82 * snp2) / r10 + 27 * sqrt77 *
674 (99 + 156 * cnt2 + 65 * cnt4) * (cnp4 * g84 + h84 * snp4) * stn4 / (512 *
675 r10) + sqrt34749 / sqrt8192 * (13 + 15 * cnt2) * (cnp6 * g86 + h86 * snp6)
676 * stn6 / r10 + 27 * sqrt715 * (cnp8 * g88 + h88 * snp8) * stn8 / (128 *
677 r10) - 27 * (35 * ct - 385 * ctn3 + 1001 * ctn5 - 715 * ctn7) * (cp * g81
678 + h81 * sp) * st / (32 * r10) + sqrt93555 * (3 * ct - 26 * ctn3 + 39 *
679 ctn5) * (cnp3 * g83 + h83 * snp3) * stn2 * st / (32 * r10) - 27 * sqrt1001
680 * (ct - 5 * ctn3) * (cnp5 * g85 + h85 * snp5) * stn4 * st / (32 * r10) +
681 27 * sqrt715 * ct * (cnp7 * g87 + h87 * snp7) * stn7 / (32 * r10);
682

```

```

683  _bt[8] = (9 * (1225 * ct + 1155 * cnt3 + 1001 * cnt5 + 715 * cnt7) * g80 * st
/
684  (1024 * r9) - 3 * sqrt35 / sqrt131072 * (105 * ct + 121 * cnt3 + 143 *
685  cnt5 + 143 * cnt7) * (cnp2 * g82 + h82 * snp2) * st / r9 - 3 * sqrt77 *
686  (138 * ct + 117 * cnt3 + 65 * cnt5) * (cnp4 * g84 + h84 * snp4) * stn3 /
687  (128 * r9) - sqrt3861 / sqrt512 * (9 * ct + 5 * cnt3) * (cnp6 * g86 + h86
688  * snp6) * stn5 / r9 - 3 * sqrt715 * ct * (cnp8 * g88 + h88 * snp8) * stn7
689  / (16 * r9) - 3 * (35 * cnt2 + 154 * cnt4 + 429 * cnt6 + 1430 * cnt8) * (1
690  / st) * (cp * g81 + h81 * sp) * st / (1024 * r9) - sqrt10395 * (21 + 42 *
691  cnt2 + 39 * cnt4 + 26 * cnt6) * (cnp3 * g83 + h83 * snp3) * st * st / (256
692  * r9) - 3 * sqrt1001 * (11 + 19 * cnt2 + 10 * cnt4) * (cnp5 * g85 + h85 *
693  snp5) * stn3 * st / (64 * r9) - 3 * sqrt715 * (3 + 4 * cnt2) * (1 / st) *
694  (cnp7 * g87 + h87 * snp7) * stn7 / (32 * r9)) / r;
695
696  _bp[8] = - ((1 / st) * (3 * sqrt35 / sqrt512 * (35 / 128 + cnt2 / 2 + 11 *
cnt4 /
697  32 - 143 * cnt8 / 128) * (cnp2 * h82 - g82 * snp2) / r9 + 3 * sqrt77 * (99
698  + 156 * cnt2 + 65 * cnt4) * (cnp4 * h84 - g84 * snp4) * stn4 / (128 * r9)
699  + sqrt3861 / sqrt2048 * (13 + 15 * cnt2) * (cnp6 * h86 - g86 * snp6) *
700  stn6 / r9 + 3 * sqrt715 * (cnp8 * h88 - g88 * snp8) * stn8 / (16 * r9) - 3
701  * (35 * ct - 385 * cnt3 + 1001 * cnt5 - 715 * cnt7) * (cp * h81 - g81 *
702  sp) * st / (32 * r9) + sqrt10395 * (3 * ct - 26 * cnt3 + 39 * cnt5) *
703  (cnp3 * h83 - g83 * snp3) * stn2 * st / (32 * r9) - 15 * sqrt1001 * (ct -
704  5 * cnt3) * (cnp5 * h85 - g85 * snp5) * stn4 * st / (32 * r9) + 21 *
705  sqrt715 * ct * (cnp7 * h87 - g87 * snp7) * stn7 / (32 * r9)) / r);
706
707  }
708
709
710      }
711    }
712  }
713 }
714 }
715 }
716 }
717
718  real br = 0, bt = 0, bp = 0;
719
720  for (int n = 1; n <= _order; n++)
721  { br += _br[n];
722    bt += _bt[n];
723    bp += _bp[n];
724  }
725
726  // convert to tesla (from gammas) and rotate about two axes
727  return rotate(rotate(cartesian(br / 1.0e9, bp / 1.0e9, -bt / 1.0e9),
728    90 - t * 180 / PI, Y_AXIS),
729    -p * 180 / PI, Z_AXIS);
730
731
732 }
733
734 // set up the static constants

```



```

735
736 real magfield::sqrt3 = sqrt(3);
737
738         real magfield::sqrt5 = sqrt(5);
739         real magfield::sqrt6 = sqrt(6);
740         real magfield::sqrt7 = sqrt(7);
741         real magfield::sqrt8 = sqrt(8);
742         real magfield::sqrt10 = sqrt(10);
743         real magfield::sqrt15 = sqrt(15);
744         real magfield::sqrt21 = sqrt(21);
745         real magfield::sqrt27 = sqrt(27);
746         real magfield::sqrt32 = sqrt(32);
747         real magfield::sqrt35 = sqrt(35);
748         real magfield::sqrt55 = sqrt(55);
749         real magfield::sqrt77 = sqrt(77);
750         real magfield::sqrt105 = sqrt(105);
751         real magfield::sqrt125 = sqrt(125);
752         real magfield::sqrt128 = sqrt(128);
753         real magfield::sqrt135 = sqrt(135);
754         real magfield::sqrt189 = sqrt(189);
755         real magfield::sqrt231 = sqrt(231);
756         real magfield::sqrt280 = sqrt(280);
757         real magfield::sqrt343 = sqrt(343);
758         real magfield::sqrt429 = sqrt(429);
759         real magfield::sqrt512 = sqrt(512);
760         real magfield::sqrt715 = sqrt(715);
761         real magfield::sqrt840 = sqrt(840);
762         real magfield::sqrt875 = sqrt(875);
763         real magfield::sqrt945 = sqrt(945);
764         real magfield::sqrt1001 = sqrt(1001);
765         real magfield::sqrt1029 = sqrt(1029);
766         real magfield::sqrt1120 = sqrt(1120);
767         real magfield::sqrt1155 = sqrt(1155);
768         real magfield::sqrt1512 = sqrt(1512);
769         real magfield::sqrt2048 = sqrt(2048);
770         real magfield::sqrt2079 = sqrt(2079);
771         real magfield::sqrt3003 = sqrt(3003);
772         real magfield::sqrt3773 = sqrt(3773);
773         real magfield::sqrt3861 = sqrt(3861);
774         real magfield::sqrt5145 = sqrt(5145);
775         real magfield::sqrt8192 = sqrt(8192);
776         real magfield::sqrt9317 = sqrt(9317);
777         real magfield::sqrt9856 = sqrt(9856);
778         real magfield::sqrt10395 = sqrt(10395.);
779         real magfield::sqrt11319 = sqrt(11319.);
780         real magfield::sqrt18711 = sqrt(18711.);
781         real magfield::sqrt30240 = sqrt(30240.);
782         real magfield::sqrt32768 = sqrt(32768.);
783         real magfield::sqrt34749 = sqrt(34749.);
784         real magfield::sqrt86515 = sqrt(86515.);
785         real magfield::sqrt93555 = sqrt(93555.);
786         real magfield::sqrt131072 = sqrt(131072.);
787         real magfield::sqrt446875 = sqrt(446875.);
788         real magfield::sqrt4084101 = sqrt(4084101.);

```

```
789         real magfield::sqrt4204629 = sqrt(4204629.);
790         real magfield::sqrt5872581 = sqrt(5872581.);
791         real magfield::sqrt124540416 = sqrt(124540416.);
```

LISTING III

a) VECTOR.H

Header file for class to perform vector calculations

```

1 // C++ code Copyright (C) David R. Evans G4AMJ/NQOI
2
3 #ifndef VECTORH
4 #define VECTORH
5
6 #include <iostream.h>
7 #include <math.h>
8
9 #include <defines.h>
10
11 class cartesian;
12 class spherical;
13
14 enum AXES { X_AXIS = 0, Y_AXIS, Z_AXIS };
15
16 // ----- cartesian class -----
17
18 class cartesian
19 { double _len, _x, _y, _z;
20   boolean length_guaranteed;
21
22 public:
23   cartesian(void);
24   cartesian(const double, const double, const double);
25   cartesian(const spherical&);
26   cartesian(cartesian&);
27
28   void operator=(cartesian);
29   void operator=(spherical);
30   int operator==(int n) { return (length(*this) == n); }
31   int operator!=(int n) { return (length(*this) != n); }
32   void operator*=(double);
33   void operator/=(double);
34   void operator+=(cartesian c) { *this = *this + c; }
35
36   friend double length(cartesian&);
37   friend double length(const cartesian&);
38
39   inline void x(double d) { _x = d; length_guaranteed = false; }
40   inline void y(double d) { _y = d; length_guaranteed = false; }
41   inline void z(double d) { _z = d; length_guaranteed = false; }
42
43   inline double x(void) const { return _x; }
44   inline double y(void) const { return _y; }
45   inline double z(void) const { return _z; }
46
47   cartesian operator*(const cartesian&) const;

```

```

48
49 // cartesian + cartesian
50 inline cartesian operator+(const cartesian& rhs) const
51     { return cartesian(_x + rhs._x, _y + rhs._y, _z + rhs._z); }
52
53 // cartesian - cartesian
54 inline cartesian operator-(const cartesian& rhs) const
55     { return cartesian(_x - rhs._x, _y - rhs._y, _z - rhs._z); }
56
57 // -cartesian
58 inline cartesian operator-(void) const
59     { return cartesian(-_x, -_y, -_z); }
60
61 // cartesian / double
62 inline cartesian operator/(const double f) const
63     { return cartesian(_x / f, _y / f, _z / f); }
64
65 friend ostream& operator<<(ostream&, const cartesian&);
66 };
67
68 // ----- common functions -----
69
70 inline cartesian operator*(const double d, const cartesian& c)
71     { return cartesian(d * c.x(), d * c.y(), d * c.z()); }
72
73 inline cartesian operator*(const cartesian& c, const double d)
74     { return d * c; }
75
76 inline double x(const cartesian& c)
77     { return c.x(); }
78
79 inline double y(const cartesian& c)
80     { return c.y(); }
81
82 inline double z(const cartesian& c)
83     { return c.z(); }
84
85 // _dot(cartesian, cartesian) -- the underline is necessary to
86 // avoid namespace collision with the dot class
87 inline double _dot(const cartesian& v1, const cartesian& v2)
88     { return v1.x() * v2.x() + v1.y() * v2.y() + v1.z() * v2.z(); }
89
90 // cross(cartesian, cartesian)
91 inline cartesian cross(const cartesian& v1, const cartesian& v2)
92     { return v1 * v2; }
93
94 // unit(cartesian)
95 inline cartesian unit(const cartesian& c)
96     { return (c / length(c)); }
97
98 // orthogonal(cartesian)
99 inline cartesian orthogonal(const cartesian& c)
100     { return cartesian(y(c) - z(c), z(c) - x(c), x(c) - y(c)); }
101

```

```

102 cartesian rotate(const cartesian& c, const double deg, AXES = Z_AXIS);
103
104 // ----- spherical class -----
105
106 class spherical
107 { double _r, _theta, _phi;
108
109 public:
110     spherical(void);
111     spherical(const double, const double, const double);
112     spherical(const cartesian&);
113     spherical(spherical&);
114
115     void operator=(spherical);
116     void operator=(cartesian);
117
118     // spherical += spherical
119     inline void operator+=(const spherical& s)
120     { *this = *this + s; }
121
122     // spherical *= double
123     inline void operator*=(double d) { _r *= d; }
124
125     // spherical /= double
126     inline void operator/=(double d) { _r /= d; }
127
128     inline double latitude(void) const
129     { return (90 - t() * 180 / PI); }
130
131     inline double longitude(void) const
132     { return p() * 180 / PI; }
133
134     inline void r(double d) { _r = d; }
135     inline void t(double d) { _theta = d; }
136     inline void p(double d) { _phi = d; }
137
138     inline double r(void) const { return _r; }
139     inline double t(void) const { return _theta; }
140     inline double p(void) const { return _phi; }
141
142     inline spherical operator*(const spherical& s) const
143     { return cross(*this, s); }
144
145     inline spherical operator+(const spherical& s) const
146     { return (const cartesian)(*this) + (const cartesian)s; }
147
148     inline spherical operator-(const spherical& s) const
149     { return (const cartesian)(*this) - (const cartesian)s; }
150
151     inline spherical operator-(void)
152     { return spherical(_r, PI - _theta, PI + _phi); }
153
154     friend ostream& operator<<(ostream&, const spherical&);
155 };

```

```

156
157 // ----- common functions -----
158
159 inline double r(const spherical& s)
160 { return s.r(); }
161
162 inline double t(const spherical& s)
163 { return s.t(); }
164
165 inline double p(const spherical& s)
166 { return s.p(); }
167
168 // unit(spherical)
169 inline spherical unit(const spherical& s)
170 { return spherical(1.0, t(s), p(s)); }
171
172 // ----- external functions -----
173
174 extern istream& operator>>(istream&, spherical&);
175 extern istream& operator>>(istream&, cartesian&);
176 extern double angle(const cartesian&, const cartesian&);
177
178 #endif

```

LISTING III
b) VECTOR.CC
Class to perform vector calculations

```

1 // C++ code Copyright (C) David R. Evans G4AMJ/NQOI
2
3 #include <vector.h>
4
5 // ***** cartesian
6
7 // generic constructor
8 cartesian::cartesian(void)
9 { _len = _x = _y = _z = 0;
10   length_guaranteed = true;
11 }
12
13 // constructor with (double, double, double)
14 cartesian::cartesian(const double tx, const double ty, const double tz)
15 { _x = tx;
16   _y = ty;
17   _z = tz;
18   _len = 0;
19   length_guaranteed = false;
20 }
21
22 // constructor with spherical
23 cartesian::cartesian(const spherical& s)
24 { _x = r(s) * sin(t(s)) * cos(p(s));
25   _y = r(s) * sin(t(s)) * sin(p(s));
26   _z = r(s) * cos(t(s));
27   _len = r(s);
28   length_guaranteed = true;
29 }
30
31 cartesian::cartesian(cartesian& c)
32 { _len = c._len;
33   _x = c._x;
34   _y = c._y;
35   _z = c._z;
36   length_guaranteed = c.length_guaranteed;
37 }
38
39 void cartesian::operator=(cartesian c)
40 { _len = c._len;
41   _x = c._x;
42   _y = c._y;
43   _z = c._z;
44   length_guaranteed = c.length_guaranteed;
45 }
46
47 // cartesian = spherical
48 void cartesian::operator=(spherical s)
49 { _x = r(s) * sin(t(s)) * cos(p(s));

```

```

50  _y = r(s) * sin(t(s)) * sin(p(s));
51  _z = r(s) * cos(t(s));
52  _len = r(s);
53  length_guaranteed = true;
54  }
55
56  // cartesian *= double
57  void cartesian::operator*=(double d)
58  { _len *= d;
59    _x *= d;
60    _y *= d;
61    _z *= d;
62  }
63
64  // cartesian /= double
65  void cartesian::operator/=(double d)
66  { _len /= d;
67    _x /= d;
68    _y /= d;
69    _z /= d;
70  }
71
72  // length(const cartesian)
73  double length(const cartesian& cc)
74  { if (cc.length_guaranteed) then
75    return cc._len;
76    else
77    // it would be nice to have ~const here!
78    { cartesian* non_const = &cc;    // non-standard, non-portable kludge
79      non_const->_len = sqrt(cc.x() * cc.x() + cc.y() * cc.y() + cc.z() * cc.z());
80      non_const->length_guaranteed = true;
81
82      return sqrt(cc._x * cc._x + cc._y * cc._y + cc._z * cc._z);
83    }
84  }
85
86  // length(cartesian)
87  double length(cartesian& param)
88  { if (!(param.length_guaranteed)) then
89    { param._len = sqrt(param._x * param._x +
90                        param._y * param._y +
91                        param._z * param._z);
92      param.length_guaranteed = true;
93    }
94    return param._len;
95  }
96
97  // cartesian * cartesian
98  cartesian cartesian::operator*(const cartesian& rhs) const
99  { return cartesian(_y * rhs._z - rhs._y * _z,
100                   -(_x * rhs._z - rhs._x * _z),
101                   _x * rhs._y - rhs._x * _y);
102  }
103

```



```

104 // output the components of a cartesian
105 ostream& operator<<(ostream& st, const cartesian& v)
106 { st << "cartesian : x = " << x(v) << " y = " << y(v) << " z = " << z(v) << "
length = " << length(v) << "\n";
107     return st;
108 }
109
110 // input the components of a cartesian
111 istream& operator>>(istream& st, cartesian& v)
112 { double tx, ty, tz;
113     st >> tx >> ty >> tz;
114     v = cartesian(tx, ty, tz);
115     return st;
116 }
117
118 // angle(cartesian, cartesian)
119 double angle(const cartesian& c1, const cartesian& c2)
120 { double param = _dot(c1, c2) / (length(c1) * length(c2));
121     param = MIN(1.0, param);
122     param = MAX(-1.0, param);
123     return acos(param);
124 }
125
126 // rotate about an axis
127 cartesian rotate(const cartesian& c, const double deg, AXES axis)
128 { const double rad = deg * PI / 180,
129     cr = cos(rad),
130     sr = sin(rad);
131     switch (axis)
132     { case X_AXIS : return cartesian(c.x(),
133                                     c.y() * cr + c.z() * sr,
134                                     -c.y() * sr + c.z() * cr);
135       case Y_AXIS : return cartesian(c.x() * cr - c.z() * sr,
136                                     c.y(),
137                                     c.x() * sr + c.z() * cr);
138       case Z_AXIS : return cartesian(c.x() * cr + c.y() * sr,
139                                     -c.x() * sr + c.y() * cr,
140                                     c.z());
141     }
142 }
143
144 // ***** spherical
145
146 // generic spherical constructor
147 spherical::spherical(void) { _r = _theta = _phi = 0; }
148
149 // spherical constructor with (double, double, double) (angles in rad)
150 spherical::spherical(const double r, const double theta, const double phi)
151 { _r = r;
152     _theta = theta;
153     _phi = phi;
154
155 // IMPORTANT NOTE -- we do not constrain the values of the private
156 // data members while the object is being built. This is because

```

```

157 // we may simply be using spherical objects as a way to do
158 // spherical arithmetic (e.g. r, theta, phi may be component
159 // lengths of a vector). If data constraint is required then build
160 // the spherical object from a cartesian
161
162 }
163
164 // spherical constructor (cartesian)
165 spherical::spherical(const cartesian& c)
166 { _r = length(c);
167   _phi = (x(c) ? atan(y(c) / x(c)) : (y(c) > 0 ? PI / 2 : -PI / 2));
168   if (x(c) < 0) then
169     _phi += PI;
170   while (_phi < 0)
171     { _phi += 2 * PI; }
172   _phi = fmod(_phi, 2 * PI);
173   _theta = (z(c) ? atan(sqrt(_r * _r - z(c) * z(c)) / z(c)) : PI / 2);
174   while (_theta < 0)
175     { _theta += PI; }
176   _theta = fmod(_theta, PI);
177 }
178
179 // ZTC-required copy-initialiser
180 spherical::spherical(spherical& s)
181 { _r = s._r;
182   _theta = s._theta;
183   _phi = s._phi;
184 }
185
186 // explicit equality operator
187 void spherical::operator=(spherical s)
188 { _r = s._r;
189   _theta = s._theta;
190   _phi = s._phi;
191 }
192
193 // spherical = cartesian
194 void spherical::operator=(cartesian c)
195 { _r = length(c);
196   _phi = (x(c) ? atan(y(c) / x(c)) : (y(c) > 0 ? PI / 2 : -PI / 2));
197   if (x(c) < 0) then
198     _phi += PI;
199   while (_phi < 0) { _phi += 2 * PI; }
200   _phi = fmod(_phi, 2 * PI);
201   _theta = (z(c) ? atan(sqrt(_r * _r - z(c) * z(c)) / z(c)) : PI / 2);
202   while (_theta < 0) { _theta += PI; }
203   _theta = fmod(_theta, PI);
204 }
205
206 // output the components of a spherical
207 ostream& operator<<(ostream& st, const spherical& v)
208 { st << "spherical : r = " << r(v) << " t = " << t(v) * 180 / PI << " p = "
209   << p(v) * 180 / PI << "\n";
210   return st;

```

```

211 }
212
213 // input the components of a spherical
214 istream& operator>>(istream& st, spherical& v)
215 { double r, t, p;
216     st >> r >> t >> p;
217     t *= PI / 180;
218     p *= PI / 180;
219     v.r(r);
220     v.t(t);
221     v.p(p);
222     return st;
223 }

```

LISTING IV WIREMODEL.CC

Program to produce the locus of magnetic field lines

```

1  // C++ code Copyright (C) David R. Evans G4AMJ/NQOI
2
3  // produce, on stdout, a wiremodel of magnetic field
4
5  #include <defines.h>
6  #include <magfield.h>
7  #include <vector.h>
8
9  #include <math.h>
10 #include <iostream.h>
11 #include <stdlib.h>
12 #include <sys/time.h>
13 #include <sys/resource.h>
14
15 // here define either URANUS or NEPTUNE
16 #undef URANUS
17 #define NEPTUNE
18
19 // if ORDER3 is defined, then uses 08 instead of full I8E1 44ev
20 #undef ORDER3
21
22 #ifdef URANUS
23 const float ellipticity = 0.0229;
24 #else
25 const float ellipticity = 0.017;
26 #endif
27
28 magfield* field;
29
30 // define matherr to abort
31 int matherr(struct exception *x)
32 { cout << "MATH ERROR\n";
33   exit(1);
34 }
35
36 // move a distance along a field line
37 cartesian along_field_line(const cartesian& p, const int direction,
38                           const double scale)
39 { const cartesian p1 = p + direction * scale * unit(field->field(p));
40   const cartesian pt = p1 - direction * scale * unit(field->field(p1));
41   const cartesian delta_p = pt - p1;
42   return p1 - 0.5 * delta_p;
43 }
44
45 main(void)
46 {
47

```

```

48 // build the correct field model
49 #ifdef URANUS
50     const int order = 2,
51           rank = 2;
52
53 // how many "g"s and "h"s are there?
54     int n_coeffs = 0;
55     for (int n = 1; n <= order; n_coeffs += ++n) ;
56
57     int32 * g, * h;
58
59     heap_check(g = new int32 [n_coeffs]);
60     heap_check(h = new int32 [n_coeffs]);
61
62 // specify the model coefficients
63     h[0] = 0; h[2] = 0;
64
65     g[0] = 11893;
66     g[1] = 11579;
67     h[1] = -15684;
68     g[2] = -6030;
69     g[3] = -12587;
70     g[4] = 196;
71     h[3] = 6116;
72     h[4] = 4759;
73
74     heap_check(field = new magfield(order, g, h));
75 #endif
76
77 #ifdef NEPTUNE
78     const int order = 8,
79           rank = 8;
80
81 // how many "g"s and "h"s are there?
82     int n_coeffs = 0;
83     for (int n = 1; n <= order; n_coeffs += ++n) ;
84
85     int32 * g, * h;
86
87     heap_check(g = new int32 [n_coeffs]);
88     heap_check(h = new int32 [n_coeffs]);
89
90 // specify the model coefficients
91     h[0] = 0; h[2] = 0; h[5] = 0; h[9] = 0;
92     h[14] = 0; h[20] = 0; h[27] = 0; h[35] = 0;
93
94 // n = 1
95     g[0] = 9732;
96     g[1] = 3220;                h[1] = -9889;
97
98 // n = 2
99     g[2] = 7448;
100    g[3] = 664;                h[3] = 11230;
101    g[4] = 4499;                h[4] = -70;

```

```

102
103
104 // n = 3
105     g[5] = -6592;
106     g[6] = 4098;
107     g[7] = -3581;
108     g[8] = 484;
109
110 // n = 4
111     g[9] = 2243;
112     g[10] = 557;
113     g[11] = 3099;
114     g[12] = -1287;
115     g[13] = -5073;
116
117 // n = 5
118     g[14] = -202;
119     g[15] = -229;
120     g[16] = 526;
121     g[17] = -2846;
122
123     g[18] = -1425;
124     g[19] = -2835;
125
126 // n = 6
127     g[20] = -2175;
128     g[21] = -466;
129     g[22] = -1269;
130     g[23] = -2233;
131     g[24] = -887;
132     g[25] = -496;
133     g[26] = 755;
134
135 // n = 7
136     g[27] = 1671;
137     g[28] = 1678;
138     g[29] = 1625;
139     g[30] = 2157;
140     g[31] = -483;
141     g[32] = 1873;
142     g[33] = 584;
143     g[34] = 664;
144
145 // n = 8
146     g[35] = -689;
147     g[36] = 238;
148     g[37] = -90;
149     g[38] = -1304;
150     g[39] = 311;
151     g[40] = -367;
152     g[41] = -249;
153     g[42] = 1333;
154     g[43] = -1239;
155
156     h[6] = -3669;
157     h[7] = 1791;
158     h[8] = -770;
159
160     h[10] = -1889;
161     h[11] = 2607;
162     h[12] = 1204;
163     h[13] = -456;
164
165     h[15] = -739;
166     h[16] = -1134;
167     h[17] = 1067;
168
169     h[18] = -1551;
170     h[19] = -1090;
171
172     h[21] = 4432;
173     h[22] = -1598;
174     h[23] = 1721;
175     h[24] = 370;
176     h[25] = -1932;
177     h[26] = 1439;
178
179     h[28] = -3159;
180     h[29] = 1862;
181     h[30] = -1120;
182     h[31] = 515;
183     h[32] = 1923;
184     h[33] = -2749;
185     h[34] = 3344;
186
187     h[36] = 1446;
188     h[37] = -79;
189     h[38] = 1043;
190     h[39] = -22;
191     h[40] = -465;
192     h[41] = 1043;
193     h[42] = -2138;
194     h[43] = 2519;

```

```

156     heap_check(field = new magfield(order, g, h));
157
158     // truncate if 08
159     #ifdef ORDER3
160         field->order(3);
161     #endif
162
163 #endif
164
165     int modulo;
166     double dlat = 160;
167     for (double dlong = 0; dlong < 360; dlong += 4)
168     { spherical pos(1, dlat * PI / 180, dlong * PI / 180);
169
170 // move so that measurements are (approximately) wrt the dipole model
171 #ifdef URANUS
172     pos = rotate(pos, 45.8, Y_AXIS);      // 105.2 | 45.8 (day | night)
173     pos = rotate(pos, 227.5, Z_AXIS);     // 47.7 | 227.5
174 #endif
175 #ifdef NEPTUNE
176     pos = rotate(pos, 138, Y_AXIS);       // 138 | 55
177     pos = rotate(pos, 57, Z_AXIS);       // 57 | 278
178 #endif
179
180 //     cerr << dlat << " " << dlong << "\n" << pos;
181     while ((r(pos) > 0.999) && (r(pos) < 20))
182     { cout << "0 " << r(pos) << " " << t(pos) * 180 / PI << " " << p(pos) * 180
/ PI << "\n";
183 // for uranus, +1 goes with first pair
184     pos = along_field_line(pos, +1, 0.01);    // +1 | -1
185     }
186 }
187 }
188

```

LISTING V
PHOTO3.CC

Program to produce plot wire models

```
1 // C++ code Copyright (C) David R. Evans G4AMJ/NQOI
2
3 // takes the output of the file wiremodel on stdin and produces a
4 // picture of the wire model on the laserprinter.
5
6 // The output format of wiremodel is the same as quadnn.
7
8 #include <DREstring.h>
9 #include <gf_lw.h>
10 #include <surface.h>
11 #include <surf_lw.h>
12 #include <vector.h>
13
14 #include <stdio.h>
15 #include <math.h>
16 #include <signal.h>
17 #include <ctype.h>
18 #include <iostream.h>
19
20 // define matherr to abort
21 int matherr(struct exception *x)
22 { cout << "MATH ERROR\n";
23   exit(1);
24   return 1;
25 }
26
27 cartesian obs(100, 100, 100), OX, XO, OI(0, 0, 1), perp_OXOI;
28
29 double zoom = 1.0;
30
31 DREstring file_name;
32
33 laserwriter* pad;
34
35 spherical screen_posn(spherical);
36 void plot_proc(void),
37 posn_proc_1(void);
38
39 /*****laser_proc*****/
40 void laser_proc(void)
41 { laserwriter lw;
42   pad = &lw;
43   plot_proc();
44   // lw.print("lw");
45   lw.detach(cout);
46 }
47
48 /*****plot_proc*****/
```



```

49 void plot_proc(void)
50 { pad->clear();
51   spherical sobs(obs);
52   *pad << text(pad->width() / 10, pad->height() / 10,
53               pad->height() / 30,
54               DREstring(t(sobs) * 180 / PI, 3.2) + ", " +
55               DREstring(p(sobs) * 180 / PI, 3.2));
56
57   /* generate the "planet" */
58   int planet_radius = (int)(pad->height() / 2 * zoom *
59                             asin(1 / length(obs)) / PI);
60   int xc = pad->width() / 2;
61   int yc = pad->height() / 2;
62
63   // we force the radius to be 100
64   zoom = 100 / (pad->height() / 2 * zoom * asin(1 / length(obs)) / PI);
65   planet_radius = (int)(pad->height() / 2 * zoom *
66                         asin(1 / length(obs)) / PI); // should be 100
67
68
69   *pad << Circle(xc, yc, planet_radius);
70
71   // define coordinate system
72   // origin = 0 = centre of planet
73   // X = observer
74   // S = point to be plotted
75   // I = (0, 0, 1) [defined to be vertically up on the screen]
76   // L = point st angle ILS is the angle between planes OXI and OXS
77
78   OX = obs;
79   XO = -OX;
80   perp_OXOI = cross(OX, OI);
81
82   for (int latitude = -90; latitude <= 90; latitude += 30)
83   { for (int longitude = 0; longitude <= 359; longitude++)
84     { if (abs(latitude) == 90) then longitude = 359;
85       spherical ll = screen_posn(spherical(1.001, (90 - latitude) * PI / 180,
86                                             longitude * PI / 180));
87       if (ll.r()) then
88         *pad << dot(xc + (int)(r(ll) * cos(t(ll))),
89                   yc - (int)(r(ll) * sin(t(ll))));
90     }
91   }
92   spherical source;
93   double alpha;
94   do
95   { cin >> alpha >> source;
96     spherical screen = screen_posn(source);
97     double distance = length(obs - source) - length(obs);
98     int dot_size = (int)((2 - distance) * 2 + 1.5);
99     dot_size = MIN(dot_size, 3);
100    dot_size = MAX(dot_size, 1);
101    if (screen.r()) then
102      *pad << dot(xc + (int)(r(screen) * cos(t(screen))),

```

```

103             yc = (int)(r(screen) * sin(t(screen))), dot_size);
104     } while (!cin.eof());
105 }
106
107 /***** print_proc *****/
108 void print_proc(void)
109 { pad->print("lw");
110   //pad->print(cout);
111 }
112
113 /***** screen_posn *****/
114 spherical screen_posn(spherical s)
115 { cartesian OS = s,
116   XS = XO + OS;
117
118   // point on screen is at R, T wrt centre of screen
119   const double R = pad->height() / 2 * zoom * angle(XO, XS) / PI,
120     plane_angle = angle(perp_OXOI, cross(OX, OS));
121
122   // which side of the central meridian?
123   const int side = SIGN(_dot(OS, perp_OXOI));
124   double T = PI / 2 + side * plane_angle;
125   while (T < 0) T += 2 * PI;
126   T = fmod(T, 2 * PI);
127
128   boolean hidden = (length(OS) <= 1);
129   double angDXO = angle(XS, XO);
130   double param = length(OX) * sin(angDXO);
131   param = MIN(param, 1.0);
132   param = MAX(-1.0, param);
133   double angODX = asin(param);
134   if (angDXO < PI / 2) then
135     angODX = PI - angODX;
136   double angDOX = PI - angODX - angDXO;
137   double lenXD = sin(angDOX)/sin(angDXO);
138
139   hidden = hidden || ((angle(XO, XS) < asin(1 / length(OX))) &&
140     ((length(XS) > lenXD)));
141   if (hidden) then
142     return spherical(0, 0, 0);
143   return spherical(R, T, 0);
144 }
145 /***** Main *****/
146 main(int argc, char** argv)
147 { if (argc != 3) then
148   { cerr << "Usage: " << argv[0] << " theta phi (degrees) < filename.\n";
149     exit(-1);
150   }
151
152   float T, P;
153   sscanf(argv[1], "%f", &T);
154   sscanf(argv[2], "%f", &P);

```

```
155  obs = spherical(100, T * PI / 180, P * PI / 180);  
156  laser_proc();  
157 }
```

REFERENCES

- Connerney, J. E. P., M. H. Acuña, and N. F. Ness, "The Magnetic Field of Uranus", *J. Geophys. Res.*, **92**, 15,329, 1987.
- Connerney, J. E. P., M. H. Acuña, and N. F. Ness, "The Magnetic Field of Neptune", *J. Geophys. Res.*, **96**, 19,023, 1991.
- Desch, M. D. and M. L. Kaiser, "Ordinary Mode Radio Emission From Uranus", *J. Geophys. Res.*, **92**, 15,211, 1987.
- Dessler, A. (ed.), *Physics of the Jovian magnetosphere*, Cambridge University Press, 1983.
- Ness, N. F., M. H. Acuña, K. H. Behannon, L. F. Burlaga, J. E. P. Connerney, R. P. Lepping, F. M. Neubauer, "Magnetic Fields at Uranus", *Science*, **233**, 85, 1986.
- Ness, N. F., M. H. Acuña, L. F. Burlaga, J. E. P. Connerney, R. P. Lepping, F. M. Neubauer, "Magnetic Fields at Neptune", *Science*, **246**, 81473, 1989.
- Roederer, J. G., *Dynamics of Geomagnetically Trapped Radiation*, Springer-Verlag, 1970.
- Wu, C. S. and L. C. Lee, "A theory of the terrestrial kilometric radiation", *Astrophys. J.* **230**, 621, 1979.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE 15 Sept 94	3. REPORT TYPE AND DATES COVERED 3 Aug 93 - 15 Sept 94, final report
4. TITLE AND SUBTITLE Non-dipolar Magnetic Field Models and Patterns of Radio Emission: Uranus and Neptune Compared			5. FUNDING NUMBERS NASA - 4801
6. AUTHOR(S) D. R. Evans			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Radiophysico, Inc. 5475 Western Ave. Boulder CO 80301			8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) NASA Code SLC Way Bergstrahl NASA Headquarters, Washington, DC 20546			10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES			
12a. DISTRIBUTION/AVAILABILITY STATEMENT			12b. DISTRIBUTION CODE
13. ABSTRACT (Maximum 200 words) The magnetic field geometries of Uranus and Neptune are examined in detail. Computer code to track low energy charged particles has been generated. Running the code against the uranian Q ₃ magnetic field model indicates that emissions previously thought to be an exotic ordinary mode emission can be more simply explained as extraordinary mode emissions from an isolated region of the inner dayside uranian magnetosphere. A similar project to examine possible particle trapping in the neptunian magnetosphere founders because of the peculiar nature of the neptunian magnetosphere.			
14. SUBJECT TERMS			15. NUMBER OF PAGES 68, including program listings
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UC	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT